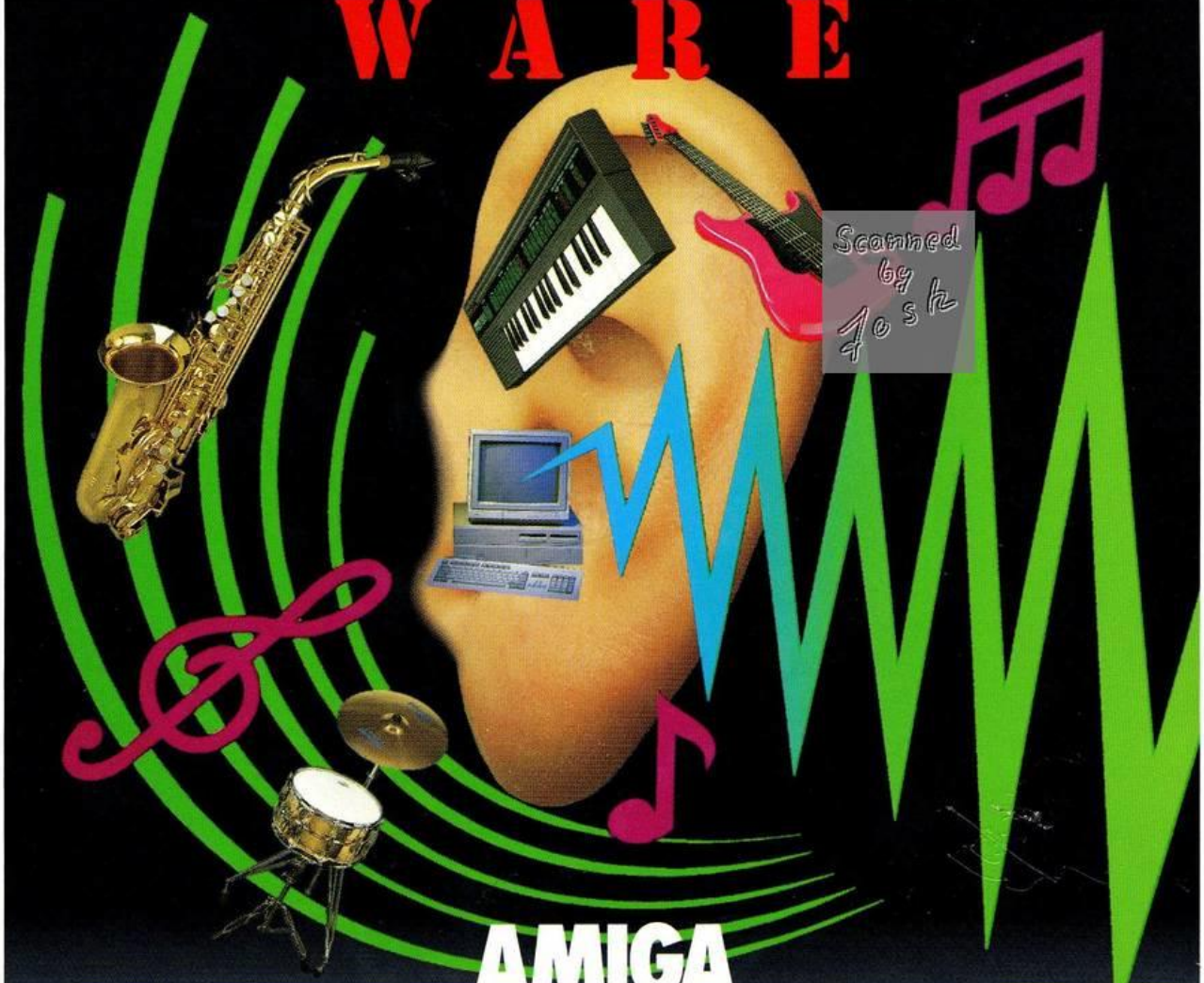
Markt&Technik

Nicolaus Wirsing

BOOK-WARE



Scanned by Jo's

AMIGA Audio Entwickler-Paket

Das Musik-Kompendium

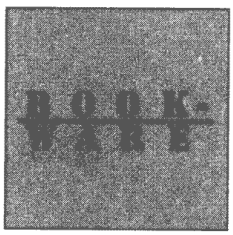
Auf Diskette enthalten:
Der komfortable SMUS-Player – voll IFF- und Sonix-kompatibel

Amiga
Audio Entwickler-Paket

Taken from Amiga-Manuals-Website

Nicolaus Wirsing

AMIGA Audio Entwickler-Paket



Das Musik-Kompendium

Markt&Technik Verlag AG

CIP-Titelaufnahme der Deutschen Bibliothek

Wirsing, Nicolaus:
AMIGA-Audio-Entwickler-Paket : das Musik-Kompendium / Nicolaus Wirsing. –
Haar bei München : Markt-und-Technik-Verl., 1991
(Bookware)
ISBN 3-89090-765-2

Die Informationen in diesem Produkt werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht.
Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt.
Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter Sorgfalt vorgegangen.
Trotzdem können Fehler nicht vollständig ausgeschlossen werden.
Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.
Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien.
Die gewerbliche Nutzung der in diesem Produkt gezeigten Modelle und Arbeiten ist nicht zulässig.

Amiga und Amiga DOS sind eingetragene Warenzeichen von Commodore-Amiga Inc., USA.
Der Gebrauch beziehungsweise die Verwendung bedarf der Erlaubnis der Schutzrechtsinhaberin.

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
94 93 92 91

ISBN 3-89090-765-2

© 1991 by Markt&Technik Verlag Aktiengesellschaft,
Hans-Pinsel-Straße 2, D-8013 Haar bei München/Germany
Alle Rechte vorbehalten
Einbandgestaltung: Grafikdesign Heinz Rauner
Dieses Produkt wurde mit Desktop-Publishing-Programmen erstellt
und auf der Linotronic 300 belichtet.
Druck: Jantsch, Günzburg
Printed in Germany

Taken from Amiga-Manuals Website

Inhaltsverzeichnis

I

Vorwort des Verlags 9

Einleitung 11

Teil A Digitale Audiotechnik

1 Grundlagen der digitalen Audiotechnik 15

1.1 Töne und Geräusche 16
1.1.1 Das Gehör – ein biologischer Spektralanalysator 16
1.1.2 Klangspektrum und Fourier-Analyse 21
1.1.3 Beschreibung akustischer Ereignisse 27

1.2 Die digitale Verarbeitung von Audiosignalen 31
1.2.1 Abtastung 33
1.2.2 Wiedergabe 54

2 Die Audiohardware des Amiga 59

2.1 Die Customchips 60
2.1.1 Paula – die D/A-Wandlerin 60
2.1.2 Agnus – die DMA-Kontrolle 62

2.2 Klangwiedergabe 64
2.2.1 Die Berechnung von Klangdaten 64
2.2.2 DMA-Betrieb 70
2.2.3 Interrupts 81
2.2.4 Modulation 89
2.2.5 Prozessor-Betrieb 95
2.2.6 Audio-Zustandsdiagramm 98

3 Systemsoftware – das Audio-Device 107

3.1 Was ist ein Device? 107
3.1.1 Messages und Ports 107

3.1.2	Device-I/O	111
3.2	Audioausgabe als Device	113
3.3	Ein Beispielprogramm zum Audio-Device	121
4	**Von der Idee zum Sound**	129
4.1	Sampling	129
4.1.1	Die Signalquelle	130
4.1.2	Störeinflüsse und deren Beseitigung	132
4.1.3	Die Aufnahme	133
4.2	Die Erstellung von Instrumenten	134
4.2.1	Korrektur der Lautstärke	135
4.2.2	Korrektur der Tonhöhe	135
4.2.3	One-Shot- und geloopte Instrumente	136
4.3	Komposition	138
4.3.1	Sonix-Tips und -Tricks	138
4.3.2	Schlagzeug-Tracks	139
4.3.3	Backing-Tracks	141
5	**MIDI**	145
5.1	Was ist MIDI?	146
5.1.1	Die Interface-Hardware	146
5.1.2	MIDI-Schaltschemata	148
5.1.3	Die MIDI-Channels (Kanäle)	151
5.1.4	Die MIDI-Modes (Betriebsarten)	152
5.1.5	Das MIDI-Datenformat	155
5.1.6	Die Unterteilung der MIDI-Befehle	156
5.1.7	Die MIDI-Befehle auf einem Blick	156
5.1.8	Die MIDI-Implementation-Chart	170
5.2	Das MIDI-Device	175
5.2.1	MIDI als Device	175
5.2.2	Die Strukturen des MIDI-Device	177
5.2.3	Funktionen und Kommandos	179
5.2.4	Lesen vom MIDI-Device	183
5.2.5	Senden über das MIDI-Device	185
5.2.6	Einschränkungen	186
5.2.7	Installation	186
5.2.8	Beispielprogramme	187

Taken from Amiga-Manuals-Website

5.3	MIDI-kompatible Musikinstrumente	220
5.3.1	Syntheseverfahren	221
5.3.2	MIDI-kompatible Synthesizer und andere Geräte	225
6	**Multitrack- und Masterrecording**	237
6.1	Effekte	237
6.2	Die Bandmaschine	242
6.2.1	Die Klangeigenschaften einer Bandmaschine	242
6.2.2	Rauschunterdrückungssysteme	244
6.2.3	Wichtige Features einer Bandmaschine	245
6.3	Synchronisation	247
6.4	Die Aufnahme	249

Teil B Das SMUSPlayer Developers Kit

1	**Einführung – die Entstehung des IFF-Formats**	255
1.1	Anforderungen an ein neues Format	255
1.2	Datentypen und Speicherbedarf	256
1.2.1	Adressenanpassung (Alignment)	256
1.2.2	Numerische Werte	256
1.2.3	Textdateien (Characters)	256
1.2.4	Identifier	257
1.3	Das Chunk-Prinzip	257
1.3.1	Identifier und Längenzähler	257
1.3.2	Der FORM-Chunk	259
1.3.3	LIST, CAT und Properties	261
2	**SMUS IFF–Simple-Musical-Score-Interchange-File-Format**	263
2.1	Definitionen	263
2.1.1	Events	263
2.1.2	Tracks	264
2.1.3	Instrumente	264
2.2	Standart-SMUS-Chunks	265
2.2.1	Der SHDR-Chunk	265

Taken from Amiga-Manuals-Website

2.2.2	Die optionalen Text-Chunks NAME, (c), AUTH, ANNO	267
2.2.3	Der INS1-Chunk	268
2.2.4	Der TRAK-Chunk	268
2.3	Programmspezifische (Non-Standard-)Chunks	276
2.4	SMUS-Schnellübersicht	276
2.4.1	Typ-Definitionen	276
2.4.2	SMUS-Standardausdrücke	279
3	**Instrumentenformate**	281
3.1	Das 8SVX-IFF-Format (8 Bit Sampled Voice)	281
3.1.1	Standard-8SVX-Chunks	282
3.1.2	FORM-S8VX-Schnellübersicht	290
4	**Die SMUSPlayer-Library**	293
4.1	Einführung	293
4.1.1	Was ist eine Library?	293
4.1.2	Die SMUSPlayer-Library	294
4.2	Die Programmierung der Library	298
4.2.1	Die Programmierung der Library in Assembler	299
4.2.2	Die Programmierung der Library in C	300
4.2.3	Die Programmierung der Library in Basic	301
4.3	Das Programm SetPacker	301
4.3.1	Der Start von SetPacker	301
4.3.2	Das Auswahlfenster	302
4.3.3	Die Zusammenstellung eines Score-Sets	304
4.4	SMUSPlayer-Library-Funktionsbeschreibung	304
4.5	Die Demoprogramme	310
4.5.1	Assembler: der CLI-Befehl Play	310
4.5.2	Beispiel in C	314
4.6	Installation	324
4.6.1	Anlegen einer Sicherheitskopie	324
4.6.2	Installation der SMUSPlayer-Library	324
4.6.3	Programmiersprachen-spezifische Installationen	324
	Stichwortverzeichnis	327

Taken from Amiga-Manuals-Website

Vorwort des Verlags

Mit Trauer und Bestürzung hat der Verlag während der Arbeit am Manuskript vom völlig unerwarteten Tod des Autors Nicolaus Wirsing erfahren müssen.

Nicolaus Wirsing zählte zu den Experten für das Thema »Computertechnik im Musikbereich« und experimentierte als Musiker besonders gern mit dem Amiga. Sein fundiertes Wissen war in unserem Hause sehr gefragt.

Der Verlag sah sich im Sinne des Autors verpflichtet, dieses umfangreiche Werk zu vollenden und einem breiten Publikum anzubieten.

Bei der Fertigstellung waren maßgeblich die Herren Heiko Knappe und Iljas Vassiliou beteiligt, die Manuskript und Software in die endgültige Form gebracht haben.

Lektorat und
Verlagsleitung

Januar 1991

Einleitung

Der Amiga ist ein Erfolgscomputer. Allen zeitlichen Vorsprüngen der jeweiligen Konkurrenzprodukte sowie der anfänglich nicht gerade offensiven Verkaufsstrategie des Herstellers zum Trotz, hat der Amiga einen Siegeszug im Home- und Semiprofessional-Computermarkt angetreten; mit ca. 40% des Gesamtumsatzes bildet er ein nicht unerhebliches wirtschaftliches Standbein der Firma Commodore. Dieser Erfolg ist wohl nicht zuletzt auf die, wenn man den Stand der Technik berücksichtigt, fast genial erscheinende Kombination von Leistungsdaten zurückzuführen, die mittlerweile auch auf dem Software-Markt die ihr gebührenden Früchte trägt. Vor allem aber die zunehmende Verwendung in professionellen Bereichen wie Werbung, Musik- und Videoproduktion zeigt das enorme Potential des Amiga hinsichtlich seiner Sound- und Grafikfähigkeiten.

Um diese große Leistungspalette zu erzielen, zog Commodore Amiga alle Register der modernen Mikroelektronik: der Mikroprozessor 68000 von Motorola, der intern eine 32-Bit-Verarbeitung erlaubt und dessen Nachfolgemodelle den späteren Amiga-Versionen die Leistungsfähigkeit einer besseren Workstation geben sollen; dieser und das ganze Orchester von Spezialchips, die in eindrucksvoller Weise zusammenwirken, gewährleisten die vielfältigen Funktionen, welche den Amiga so weit über das Niveau seiner Preisklasse heben.

Diese Fülle an Features und Möglichkeiten bleibt natürlich nicht ohne Folgen auf die Komplexität der Programme, die sich ihrer bedienen. Die Ansprüche, die sowohl an den Programmierer als auch an den Anwender gestellt werden, steigen bei hardwarebezogenen Programmen, wie z.B. Mal- und Klangverarbeitungsprogrammen, mit den zunehmenden Fähigkeiten der Hardware und Systemsoftware stark an.

Speziell im Audiobereich ist die große Menge an Information, die zum Verständnis der Vorgänge in einem Computer wie dem Amiga notwendig ist, nur einigermaßen akzeptabel zu bewältigen, wenn man sich diese nicht erst in mühseliger Kleinarbeit zusammenkratzen muß. Da der Amiga ein nach dem Sampling-Prinzip arbeitendes Klangwiedergabesystem besitzt, sind in diesem Zusammenhang auch die Grundlagen der digitalen Audiotechnik von großem Interesse, welche allgemein in der Klangtechnik immer größere Bedeutung erlangt. Dieses Buch soll Ihnen einen Weg zu der von Ihnen gewünschten Audio-Anwendung zeigen, ohne dabei hohe Anforderungen an Ihr mathematisches und physikalisches Verständnis zu stellen. Dem Programmier-Profi sowie dem musikinteressierten Anwender bietet es jedoch die Menge an Hintergrundinformation, die es ihm gestattet, sämtliche Fähigkeiten des Systems voll auszunutzen.

Der Amiga enthält, wie erwähnt, ein System zur Wiedergabe digitaler Daten als analoge elektrische Spannung, welche in Klang (z.B. durch einen Lautsprecher) umgesetzt werden können. Die Menge an Klangdaten, und damit auch die Abspielzeit des jeweiligen Sounds, ist

jedoch durch die Speicherkapazität des Computers begrenzt. Um diese Kapazität optimal auszunutzen, ist es wünschenswert, Klangdaten einer mehrfachen Verwendung zuzuführen, d.h. deren Ausgabe gegebenenfalls mehrfach zu wiederholen oder sie gar als »Instrumente« einzusetzen. Zu diesem Zweck bedienen sich die meisten Programme, die dies erlauben und so die Erstellung von »Kompositionen« gestatten, des sogenannten SMUS-Formats (wie z.B. Sonix, Deluxe Music Construction Set, Music Studio, Instant Music etc.). In diesem Buch soll der Einsatz dieser Programme anhand einiger Beispiele näher erläutert werden.

An MIDI, der digitalen Schnittstelle zwischen modernen elektronischen Musikinstrumenten und Computern, kommt heute kein musikbegeisterter Amiga-Anwender mehr vorbei. Aus diesem Grunde befaßt sich der erste Teil des vorliegenden Bookware-Pakets sehr intensiv mit der aktuellen MIDI-Implementation und stellt dem interessierten Programmierer ein leistungsfähiges MIDI-Device zur Verfügung, welches die Ansteuerung angeschlossener MIDI-Geräte über ein MIDI-Interface ermöglicht.

Der zweite Teil des Bookware-Pakets beinhaltet die SMUSPlayer.Library, welche es Ihnen erlaubt, Ihre Kompositionen unabhängig von den jeweiligen Kompositionsprogrammen wiedergeben zu lassen. Mit ihrer Hilfe können Sie bequem und ohne den üblichen Programmieraufwand selbstkomponierte Lieder innerhalb Ihrer eigenen Programme abspielen lassen, sei es nun zur Untermalung einer Präsentationsgrafik, einer Videoanimation oder eines Spieles. Speziell für Programmierer von Spielen wurde eine Möglichkeit geschaffen, IFF- oder Sonix- Instrumente neben den Stimmen des abzuspielenden Liedes separat wiederzugeben. Überdies können eigene Programme mit dem Ablauf eines Liedes synchronisiert werden. Anhand von reichhaltig dokumentierten Beispielprogrammen werden die Aufruf-Techniken, die zum Betrieb des SMUSPlayers benötigt werden, eingehend erläutert.

Die Zielsetzung dieses Buches ist es, den musikinteressierten Anwender wie den Programmierer mit allen Informationen, die zur vollständigen Ausnutzung der Soundfähigkeiten des Amiga erforderlich sind, zu versorgen. Hierbei sollen sowohl die Aspekte reiner Computeranwendungen, wie Animationen und Spiele, als auch der professionellen Musikproduktion, bei der dem Computer meist die Aufgabe einer MIDI-Recorder- und Steuereinheit zufallen, gleichermaßen beleuchtet werden.

Taken from Amiga Manuals Website

Digitale Audiotechnik

Teil

A

Grundlagen der digitalen Audiotechnik

1

Die rasante technische Entwicklung der letzten fünfzig Jahre hat auch den Bereich der Klangerzeugungs- und Reproduktionstechniken zu keiner Zeit unberührt gelassen. Nachdem man in den zwanziger Jahren mit der breiten Nutzung von elektroakustischen Verfahren begonnen hatte, ließen sich diese bald zu brauchbaren Werkzeugen der Instrumentierung, Aufnahme und Wiedergabe von Musik machen, was vor allem in der Unterhaltungsmusik einen großen Effekt zeigte. Nach dem Aufkommen zahlreicher elektroakustischer Instrumente erschien in den sechziger Jahren mit dem MOOG-Synthesizer das erste frei durchstimmbare, rein elektronische Klangerzeugungsgerät, das keiner mechanisch schwingenden Systeme mehr bedurfte. Das hier verwendete Prinzip beruhte auf elektronischen Schwingkreisen, deren Frequenz, d.h., deren Tonhöhe durch eine Spannung von außen frei bestimmt werden konnte (VCO: Voltage Controlled Oscillator). Diese Form der Tonerzeugung rechnet man den »analogen« Verfahren zu, bei denen die zu behandelnden Größen in unendlich fein abgestuften Werten vorliegen, wie z.B. auch bei Längen, Gewichten, Temperaturen oder eben elektrischen Spannungen.

Ein anderer Zweig der Elektronik war es, namentlich die Digitaltechnik, die zuerst im professionellen Bereich, dann auch im semiprofessionellen und Hobbysektor zu einer enormen Zunahme der Fähigkeiten und der Flexibilität elektronischer Musikinstrumente geführt hat. Diese bedient sich einer Darstellung der Größen in Form von ganzzahligen Werten, deren kleinste Einheit, das Bit, nur die Werte 0 und 1 annehmen kann. Hinsichtlich der Bearbeitung und Reproduktion der physikalischen Größen hat dies den entscheidenden Vorteil, daß man diesen beiden Werten, 0 und 1, nur hinreichend gut unterscheidbare Spannungswerte zuzuweisen braucht, um deren Übertragung unempfindlich gegen Störeinflüsse werden zu lassen; dagegen macht sich in der Analogtechnik jede Störung im Ergebnis einer Übertragung oder eines Kopiervorgangs bemerkbar.

Infolge der extremen Preissenkungen auf dem Markt digitaler Bausteine sowie der immer perfekteren Fertigungstechniken konnten eine Fülle von preiswerten Keyboards, Synthesizern, Schlagzeugmaschinen etc. entwickelt und angeboten werden. Die Funktionsweise der meisten Geräte, die in diesem Bereich auf dem Markt sind, dürfte wohl inzwischen ganz oder zumindest teilweise auf Prinzipien der Digitaltechnik beruhen. Bei den eigentlichen Repräsentanten dieser Technik wiederum, den Computern, macht sich der Trend bemerkbar, für Videospiele oder auch ernsthaftere Musikanwendungen immer ausgefeiltere Klangerzeugungssysteme zu integrieren.

Wohl eines der interessantesten Ergebnisse dieser Tendenz ist der Commodore Amiga. Im Vergleich zu früheren Entwicklungen wie dem Atari ST und dem Commodore 64, in welchen Klangbausteine mit frei wählbarer Frequenz und beschränkt variabler Wellenform arbeiten (was lediglich eine Umsetzung des Schwingkreis-Prinzips in eine digitale Entsprechung bedeutet,

daher die Bezeichnung DCO = Digital Controlled Oscillator), wurde der Amiga mit einer wesentlich flexibleren Soundsynthese ausgestattet. Ähnlich dem Wiedergabeverfahren bei einem CD-Player, besteht die Klangerzeugungseinheit des Amiga aus sogenannten Digital/Analog-Wandlern, also Bausteinen, die eine beliebige Folge von digitalen Zahlenwerten in analoge Spannungen und damit, unter Zuhilfenahme eines Verstärkers und eines Lautsprechers, in Klang umzuwandeln vermögen. Dieser Weg erlaubt die Wiedergabe einer nahezu unbegrenzten Fülle von Klangfarben und Geräuschen, da praktisch jedes akustische Ereignis durch eine solche Zahlenfolge dargestellt werden kann.

Die Grundlagen dieser »Sampling« genannten Technik, also der periodischen Probenentnahme, sowie deren Konsequenzen sind das Thema dieses Kapitels. Im ersten Abschnitt sollen zunächst die Voraussetzungen für die Wahrnehmung und die physikalische Beschreibung von Klängen erläutert werden, wobei vor allem die Beziehung zwischen beiden hervorgehoben werden soll. Der zweite Abschnitt behandelt die Phänomene, die bei der Bearbeitung von Musik mit digitalen Verfahren auftreten. Insbesondere die Aspekte der Sampling- und Wiedergabepraxis, wie sie für die Arbeit mit dem Amiga typisch sind, werden hier eingehend diskutiert.

Da dieses Kapitel keine unabdingbare Voraussetzung zum Umgang mit den auf dem Amiga verfügbaren Musikanwendungen darstellt, können Leser, die sich ausschließlich mit den Software-Werkzeugen beschäftigen wollen, wie z.B. Profi-Musiker, diese Passagen überspringen und sich sofort den für sie interessanten Abschnitten zuwenden. Sollten Sie jedoch vorhaben tiefer in die Techniken der Herstellung von eigenen Samples oder in die Programmierung der Audioeinheit des Amiga einzusteigen, so werden sich diese Hintergrundinformationen in der Regel als wertvoll erweisen, um die Vorgänge und Effekte bei der digitalen Klangverarbeitung besser verstehen und beherrschen zu können.

1.1 Töne und Geräusche

1.1.1 Das Gehör – ein biologischer Spektralanalysator

Lautstärke und Amplitude

Einen wesentlichen Bestandteil unserer gesamten sinnlichen Erfahrung bilden die auditiven Empfindungen, d.h. die Klänge, die wir bewußt oder weniger bewußt wahrnehmen und die bei uns Reiz oder Ablehnung, Wohlbefinden oder Unbehagen verursachen können. Während die Zusammenhänge zwischen psychischen Zuständen und Klangwahrnehmung sowie speziell des Dissonanz- und Harmonieempfindens noch nicht vollständig geklärt sind, kann man sich doch aufgrund der genauen anatomischen Kenntnisse des menschlichen Gehörapparates ein recht gutes Bild von der Funktionsweise der hierfür zuständigen Organe machen. Dasjenige physikalische Ereignis, welches die Wahrnehmung von Klang hervorruft, ist eine Schwankung des Druckes der Luft, welche sich wellenförmig ausbreitet und durch das Trommelfell und die Gehörknöchelchen an eine Flüssigkeit in der Innenohrschnecke weitergegeben wird. Hier befinden sich die eigentlichen Klangrezeptoren, Sinneszellen, die auf Strömungen der sie umgebenden Flüssigkeit reagieren und ihre Informationen an den Hörnerv weitergeben, bevor dieser in den unüberblickbaren Verästelungen des Gehirns verschwindet.

Tekno Amiga Anwendungen

Möchte man nun die Eigenschaften unserer Klangwahrnehmung untersuchen, so muß man sich die Frage stellen, welche Informationen eine solche Anordnung aus der an sie herangetragenen Botschaft ziehen kann. Als erstes sind hier wohl die Rezeptoren selbst, d.h. die Sinneshärchen an den Wänden der Innenohrschnecke, in Betracht zu ziehen. Diese reagieren bei verschiedenen Stärken der sie erreichenden Schallwellen mit unterschiedlichen Reizintensitäten, die sie an den Hörnerv weitergeben. So kann das Gehör die Größe der Druckunterschiede, sprich die Lautstärke des Schalls sehr genau bestimmen.

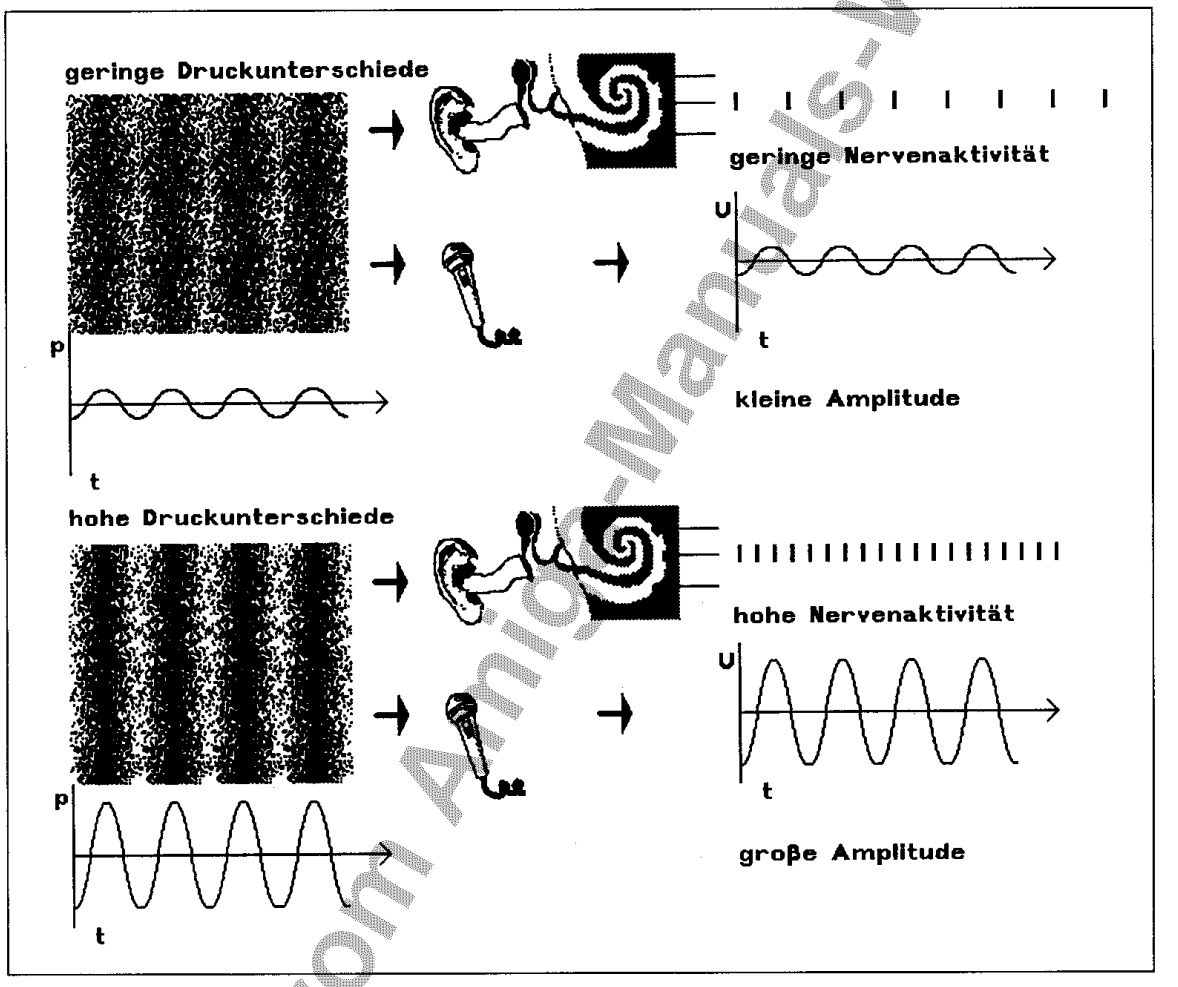


Abb. 1.1: Lautstärke und Amplitude

Da das Gehör Lautstärken vom kleinsten wahrnehmbaren Druckunterschied, 0.2 nbar (1 Nanobar = 10^{-9} bar), bis das ca. 1-Million-fache davon registrieren und verarbeiten muß, erfolgt die Empfindung von Lautstärkeverhältnissen nicht proportional zum Schalldruck, sondern logarithmisch dazu. Wäre dies nicht der Fall, so wäre für die höchste Lautstärke, im Verhältnis zur kleinsten, die millionenfache Anzahl von Nervenimpulsen erforderlich, was jedoch bei einer maximalen Feuergeschwindigkeit eines Neurons von ca. 400 Impulsen pro Sekunde vollkommen

Taken from Amiga-Manuals-Website

unmöglich ist. Daher wird die Intensität von Schall ungefähr analog dem dekadischen Logarithmus empfunden, d.h. um z.B. den subjektiven Eindruck von doppelter Lautstärke zu erreichen, muß der zehnfache Schalldruck vorliegen, für vierfache Lautstärke der hundertfache und so fort. Dies hatte unter anderem zur Folge, daß man als Einheit für Lautstärkeverhältnisse das Dezibel (dB) wählte, welches folgendermaßen definiert ist:

$$\text{dB-Wert} = 10 * \log\left(\frac{P_2}{P_1}\right)$$

wobei $\log()$ - dekadischer Logarithmus
P_1, P_2 - Leistung (Energie pro Zeit)

Formel 1

Da die Leistung proportional zum Quadrat des Schalldruckes ist, läßt sich auch schreiben:

$$\text{db-Wert} = 10 * \log\left(\frac{D_1}{D_2}\right)^2 =$$

$$= 20 * \log\left(\frac{D_1}{D_2}\right)$$

wobei $\log()$ - dekadischer Logarithmus
D_1, D_2 - Schalldruck

Formel 2

Nimmt man die untere Hörgrenze, 0.2 nbar, willkürlich als 0 dB-Wert an, so erhält man für den menschlichen Hörbereich eine absolute Skala von 0 bis ca. 120 dB(A), was die Schmerzgrenze darstellt und ungefähr der Lautstärke eines startenden Düsenflugzeugs auf 10 Meter Entfernung entspricht.

Um Klänge nun mit elektronischen Mitteln bearbeitbar zu machen, wählt man in der Audiotechnik eine andere Darstellungsform für Klang als den Schalldruck selbst. Wie in Abb. 1.1 zu sehen, läßt

sich Schall mit Hilfe eines Mikrophons in elektrische Schwingungen umwandeln. Diese sind Schwankungen einer elektrischen Spannung, die in ihrem Verlauf den Druckschwankungen des Schalls genau entsprechen, d.h. proportional zu ihnen sind. Die Einheit dB ist deshalb für Spannungen folgendermaßen definiert:

$$\text{db-Wert} = 10 * \log\left(\frac{P_1}{P_2}\right) =$$

$$= 20 * \log\left(\frac{U_1}{U_2}\right)$$

wobei
- $\log()$ - dekadischer Logarithmus
- P_1, P_2 - elektrische Leistung
- U_1, U_2 - Spannung

Formel 3

Wie beim Schall, so ist auch die elektrische Leistung proportional zum Quadrat der Spannung. Dies entspricht einer Multiplikation des Logarithmus mit dem Faktor 2.

Da sich Luftdrücke in einem weiten Bereich genau wie elektrische Spannungen verhalten, d.h. sich genau wie diese addieren und subtrahieren, ist es möglich, durch elektrische Schwingungen dargestellte Klänge in relativ einfacher Weise zu verstärken, zu verändern, zu mischen und zu reproduzieren. So ist es für das Ohr ziemlich schwierig, zu unterscheiden, ob z.B. zwei wahrgenommene Töne durch zwei verschiedene nahe beieinanderstehende Lautsprecher erzeugt werden oder lediglich durch einen Lautsprecher, an dem die Summe der beiden Spannungen anliegt. Diese Ähnlichkeit der Eigenschaften stellt eine der grundlegenden Voraussetzungen der gesamten Audiotechnik und damit auch der digitalen Audiotechnik dar, da auch digitale Darstellungsformen von Klängen meist erst aus analogen Spannungen gewonnen werden müssen.

Tonhöhe und Frequenz

Die zweite wesentliche Eigenschaft unserer auditiven Empfindung, zu der wir aus dem Aufbau des menschlichen Ohrs Rückschlüsse ziehen können, ist die Fähigkeit, Tonhöhen zu unterscheiden. Unser Gehör empfindet sowohl eine allgemeine »Helligkeit« des Tones, also die absolute Tonhöhe, als auch einen zyklisch verlaufenden Toncharakter, d.h. eine Verwandtschaft zwischen Tönen, die eine Oktave auseinanderliegen, der höhere Ton also genau die doppelte Anzahl von Schwingungen pro Zeit aufweist. Fehlt einem Menschen die letztgenannte Fähigkeit, so spricht man von sog. Tontaubheit, die das völlige Unvermögen, Musik zu genießen, zur Folge hat.

Diese Eigenschaft einer Schwingung, die Anzahl der Schwingungsdurchläufe pro Zeiteinheit, bezeichnet man als ihre Frequenz. Sie wird im allgemeinen in der Einheit Hertz (Hz) angegeben:

1 Hz = 1 Schwingung/Sekunde

Der Tonumfang z.B. eines Klaviers läßt sich in Hertz so angeben:

A2	A1	A	a	a1	a2	a3	a4
27,5	55	110	220	440	880	1760	3520 Hz

Handelt es sich um eine wellenförmige Schwingung eines Mediums wie z.B. der Luft, so ergibt sich aus der Frequenz sowie der Geschwindigkeit des Schalls die Wellenlänge, d.h. die Entfernung zwischen den Kämmen bzw. Tälern einer Welle. Auch in anderen Medien, wie auch in der Flüssigkeit der Innenohrschnecke, bilden sich bei unterschiedlichen Frequenzen verschiedene Wellenlängen aus. Diese bewirken im Innenohr durch die konische Tütenform des Schneckenganges folgenden Effekt: An manchen Stellen heben sich die ankommenden und die vom Ende

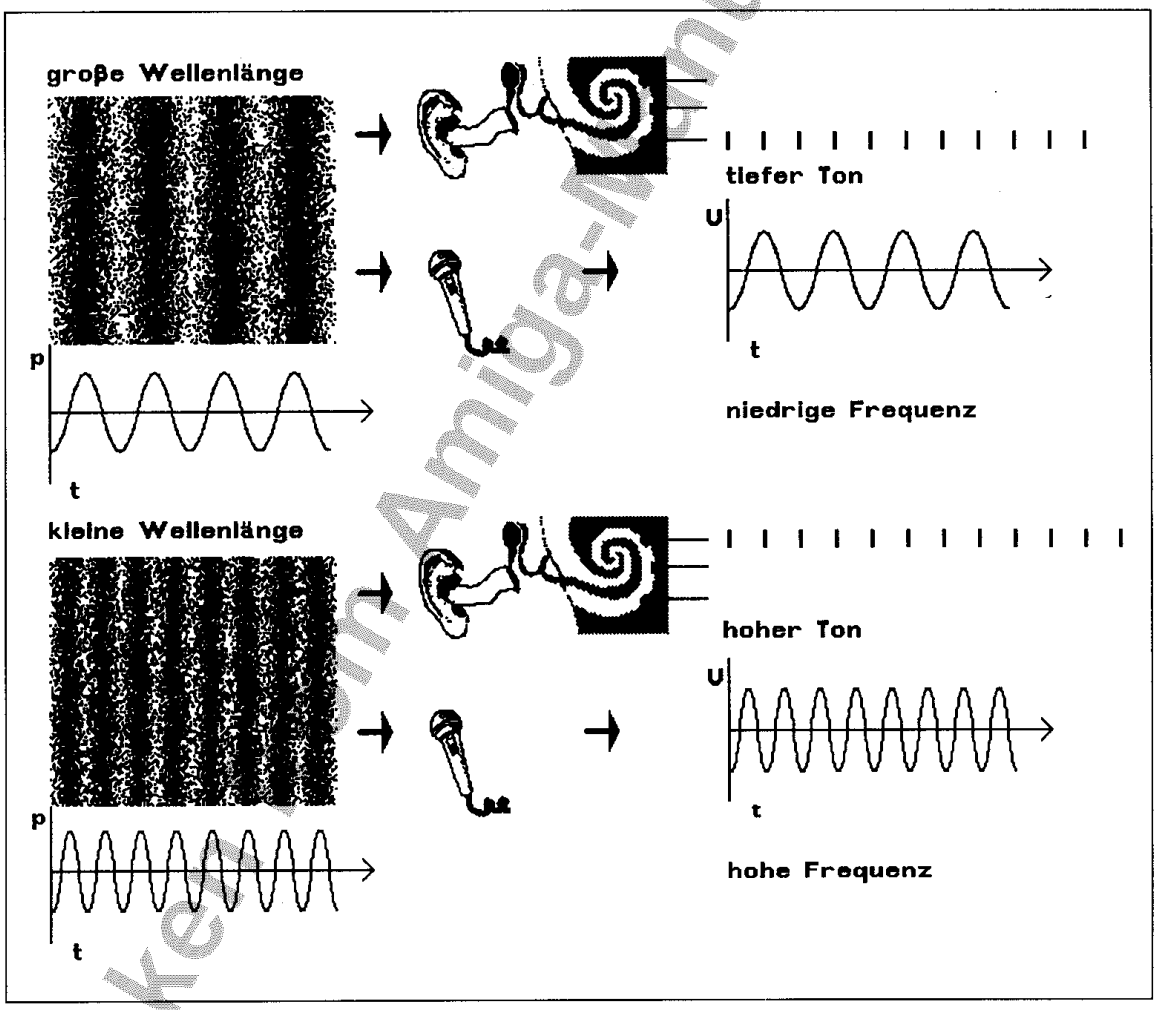


Abb. 1.2: Tonhöhe und Frequenz

Take from Amiga-Manuals-Website

des Ganges reflektierten Schallwellen gegenseitig auf, an anderen addieren sie sich zu doppelter Stärke, was zu sog. »stehenden Wellen« führt. Diese werden auch als Interferenzen bezeichnet und entstehen immer dann, wenn sich eine Schwingung mit einer entgegenkommenden gleichfrequenten überlagert. Da die räumliche Lage dieser stehenden Wellen von ihrer Frequenz abhängig ist, ergeben sich durch verschiedene Tonhöhen stets andere Orte des Schneckenganges, an denen sich Schwingungsmaxima ergeben und so die dortigen Rezeptoren gereizt werden. Anhand dieses Reizmusters analysiert das Gehör einen Klang nach den in ihm enthaltenen Wellenlängen, auch wenn sich dieser aus mehreren Tonhöhen, also Frequenzen zusammensetzt. Das Ohr ist demzufolge in der Lage, aus einem beliebigen Verlauf von Druckschwankungen eine Reihe von reinen Teilschwingungen zu isolieren, was man als die »chromatische« Eigenschaft des Gehörs bezeichnet, d.h. die Fähigkeit, einzelne Frequenzen aus einem Gemisch von Tönen herauszuhören.

Der hörbare Frequenzbereich umfaßt beim Menschen ca. 16 Hz bis 20000 Hz, wobei jedoch nicht alle Tonlagen mit gleicher Intensität wahrgenommen werden. Im untersten Bereich, um 16 Hz, werden nur relativ starke Schwingungen bemerkt, wobei die Haut als Empfängerorgan wesentlich beteiligt ist. Ab da steigt die Hörempfindlichkeit langsam an und ist im Bereich zwischen 1000 und 4000 Hz am größten, wohl unter anderem deshalb, da dies der Bereich ist, in dem die in menschlicher Sprache vorkommenden Frequenzen liegen. Von hier an sinkt die Sensitivität wieder ab, bis sie schließlich bei jungen Menschen um 18000 Hz bis 20000 Hz, bei älteren bereits um die 12000 Hz ganz verschwindet.

1.1.2 Klangspektrum und Fourier-Analyse

Im letzten Abschnitt wurde gezeigt, wie das Gehör die einzelnen Frequenzen eines Klanges voneinander trennt, eine Summe von Schwingungen also in ihre Bestandteile zerlegen kann; dies ist zur Unterscheidung von gleichzeitigen akustischen Ereignissen auch unbedingt nötig. Tatsächlich begegnen wir in der Realität so gut wie nie einer wirklich reinen Schwingung, fast alle Klänge, die wir wahrnehmen, sind aus mehreren verschiedenen Frequenzen zusammengesetzt. Hört man beispielsweise den Ton einer Geige, so nimmt man außer dem Grundton, der meist die tiefste Frequenzkomponente bildet und nach dem wir den Klang in die Tonskala einordnen, noch eine ganze Reihe von sogenannten Oberwellen wahr, durch die der Charakter des Klanges wesentlich bestimmt wird. Bei einer Geige sind diese besonders stark im Verhältnis zum Grundton, was den hellen, scharfen Klang der Geige ausmacht, im Gegensatz z.B. zu einer Flöte, deren kleiner Oberwellengehalt einen dumpfen und weichen Ton ergibt.

Diesen charakteristischen Gehalt an Oberwellen, genauer deren Frequenz und Intensität gegenüber dem Grundton, nennt man das Spektrum eines Klanges. Bei den meisten akustischen Ereignissen ändert sich dieses in ihrem Verlauf; für die folgenden Betrachtungen beschränken wir uns jedoch zuerst einmal auf Klänge, die in ihrem zeitlichen Verlauf konstant, Druck- bzw. Spannungsverlauf also genau periodisch sind.

Harmonische Schwingung und Sinusfunktion

Möchte man nun das Spektrum eines Klanges bestimmen, so stellt sich als erstes die Frage, woraus die Komponenten eines Spektrums bestehen, was eine solche »reine« Schwingung überhaupt ist. In der Physik bezeichnet man Schwingungen, die dieser Idealbedingung entsprechen, als »harmo-

Telefunken-Manuals.Net

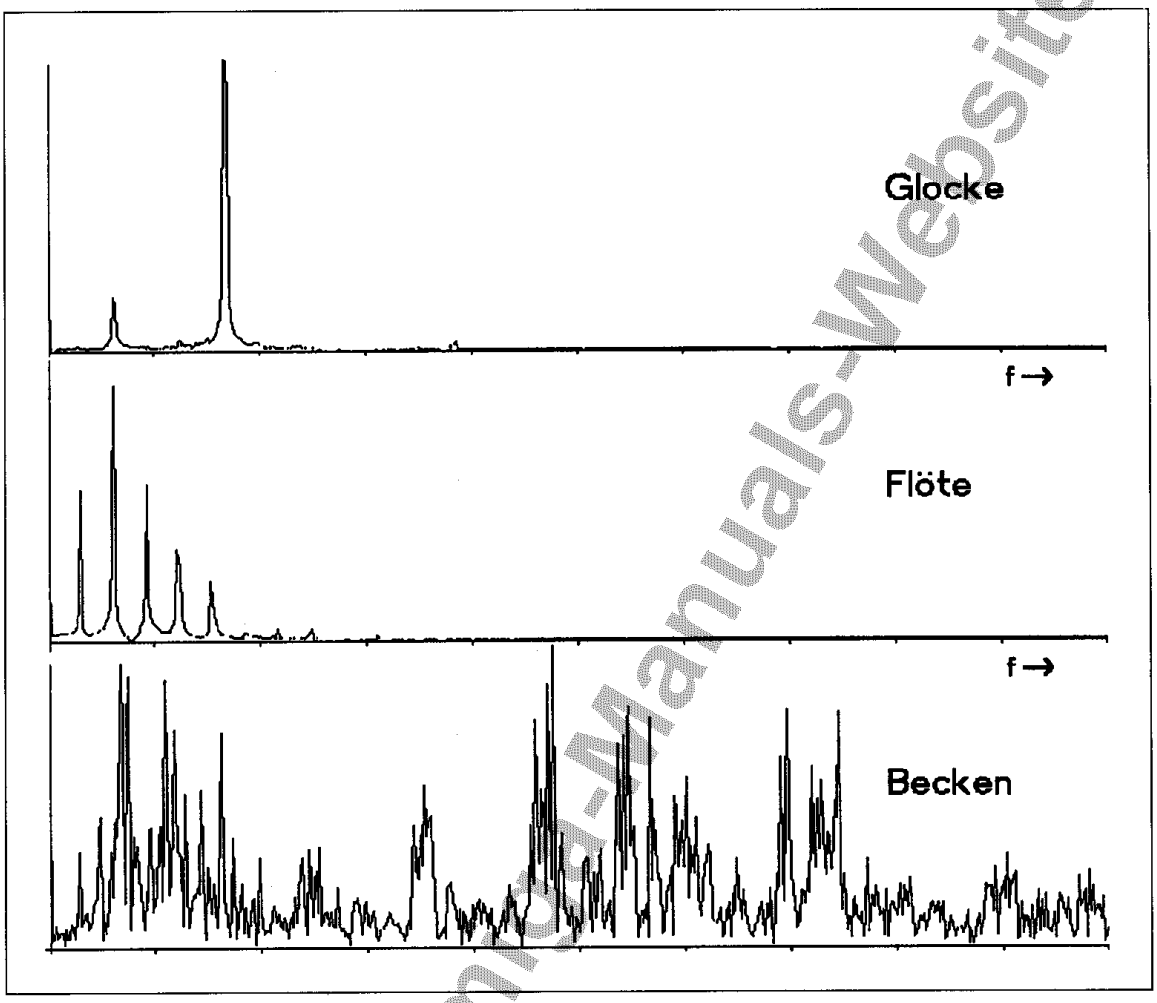


Abb. 1.3: Oberwellen von verschiedenen Instrumenten

nische« Schwingungen. Diese sind dadurch gekennzeichnet, daß die Rückstellkraft (die Kraft, die in Richtung Nullposition wirkt) linear mit der Amplitude wächst, also proportional zu ihr ist. Ein Beispiel hierfür wäre ein Pendel, bei dem ja die Kraft, die es nach unten und damit in die Mitte zieht, sich weitgehend proportional zu seiner seitlichen Auslenkung verhält. Zieht man ein Pendel aus der Mittelposition und läßt es dann los, so beschreibt es eine ähnliche Bewegung wie durch Schall zum Schwingen angeregte Luftmoleküle oder den Druckverlauf der Luft bzw. die Ausgangsspannung eines Mikrophons, das einen (reinen) Ton empfängt.

Läßt man das Pendel nun so schwingen, daß es in beide Richtungen, horizontal wie vertikal, exakt die gleiche Auslenkung erreicht, so ergibt sich eine genaue Kreisbewegung. Aus dem Kreis leitet sich auch diejenige mathematische Funktion ab, durch die die harmonische Schwingung definiert ist: Sinusfunktion. Der Sinus eines Winkels α, $\sin(\alpha)$, ergibt sich aus dem Abstand zwischen dem Schnittpunkt des einen Winkelschenkels mit dem Einheitskreis, also einem Kreis mit dem Radius 1, und der x-Achse, die den anderen Schenkel bildet. Überträgt man die Sinus-Werte sämtlicher Winkel eines Kreises nacheinander entlang einer Achse, so erhält man die für die Sinusfunktion

Taken from Amiga-Manuals-Website

typische Wellenform, die sich mit jeder vollen Umdrehung genau wiederholt; man sagt auch: sie besitzt die Periode 360 Grad oder 2.

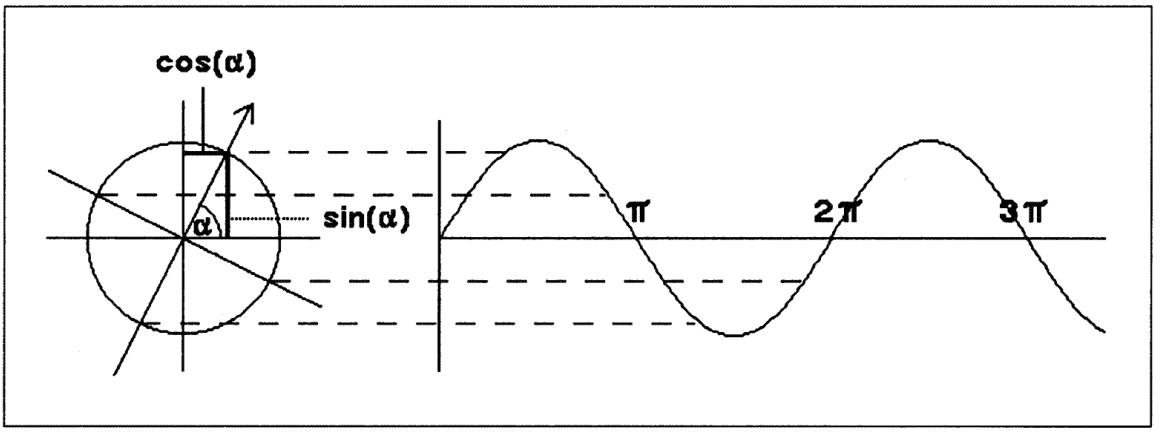


Abb. 1.4: Einheitskreis und Sinusfunktion

Diese sinusförmigen Schwingungen sind es also, nach denen unser Gehör die empfangenen Schallereignisse und ihre Druckverläufe aufteilt und analysiert. Nun stellt sich des weiteren die Frage, wie man eine solche Darstellung, zergliedert nach den einzelnen Sinusanteilen, aus einem Druck- oder Spannungsverlauf gewinnen kann, um diesen den Empfindungen des Gehörs entsprechend besser untersuchen zu können. Es muß also eine Methode gefunden werden, zu ermitteln, welche Sinuswellen mit welcher Frequenz und Amplitude zusammenaddiert werden müssen, um eine vorliegende, beliebig verlaufende Schwingung zu erhalten.

Fourier-Reihen

Hierzu gibt es in der Mathematik ein Verfahren, das als harmonische Analyse oder Fourier-Analyse bekannt ist. Bei dieser wird versucht, eine beliebige periodische Funktion f(x), in unserem Fall also die Abbildung der Zeit auf die Spannungswerte der Schwingung, durch eine Reihe, also die Summe über eine Zahlenfolge darzustellen, bei der jeder einzelne Summand eine Sinusfunktion mit jeweils anderer Periode darstellt. Bei der Fourier-Analyse sind dies die ganzzahligen Vielfachen der »Grundfrequenz« der Funktion (f, 2f, 3f...). Diese Sinusanteile werden noch mit Koeffizienten, den sogenannten Fourier-Koeffizienten, multipliziert, die ihre Gewichtung, d.h. die Intensität der einzelnen Anteile im Verhältnis zur Grundschwingung wiedergeben. Eine solche Fourier-Reihe ist im allgemeinen von folgender Gestalt:

$$f(x) = \frac{a_0}{2} + \sum_{k=1}^{\infty} \left(a_k * \cos(k*x) + b_k * \sin(k*x) \right)$$

Formel 4

Die Fourier-Koeffizienten a(k) und b(k) errechnen sich folgendermaßen:

$$a_k = \frac{1}{\pi} * \int_{-\pi}^{+\pi} (f(x) * \cos(k*x))\, dx \qquad (k = 0,1,2,...)$$

$$b_k = \frac{1}{\pi} * \int_{-\pi}^{+\pi} (f(x) * \sin(k*x))\, dx \qquad (k = 0,1,2,...)$$

Formel 5

Da sich die meisten Schwingungsfunktionen, denen wir im Audiobereich begegnen, auch allein durch entweder Sinus- oder Cosinusanteile darstellen lassen, läßt sich wegen der Symmetrie der Funktionen auch schreiben (siehe Formel 6):

Hieraus erhält man mit den Koeffizienten a(k) oder b(k) die Gewichtungen, mit denen die einzelnen Oberwellen mit den Perioden T, T/2, T/3 ... bzw. den Frequenzen f, 2f, 3f etc. multipliziert werden müssen, um die zu untersuchende Funktion f(x) zu ergeben. Je höher man k werden läßt, d.h., je mehr Teilschwingungen man in die Berechnung einbezieht, desto näher kommt das Ergebnis an f(x) heran. Man sagt auch, die Reihe konvergiert gegen f(x). Theoretisch müßte man also unendlich viele Summanden berücksichtigen, um eine Funktion exakt zu beschreiben und wirklich alle ihre Komponenten zu erhalten. In der Praxis jedoch muß man die Untersuchung natürlich nach endlich vielen Gliedern abbrechen und versuchen, eine hinreichende Annäherung zu erzielen.

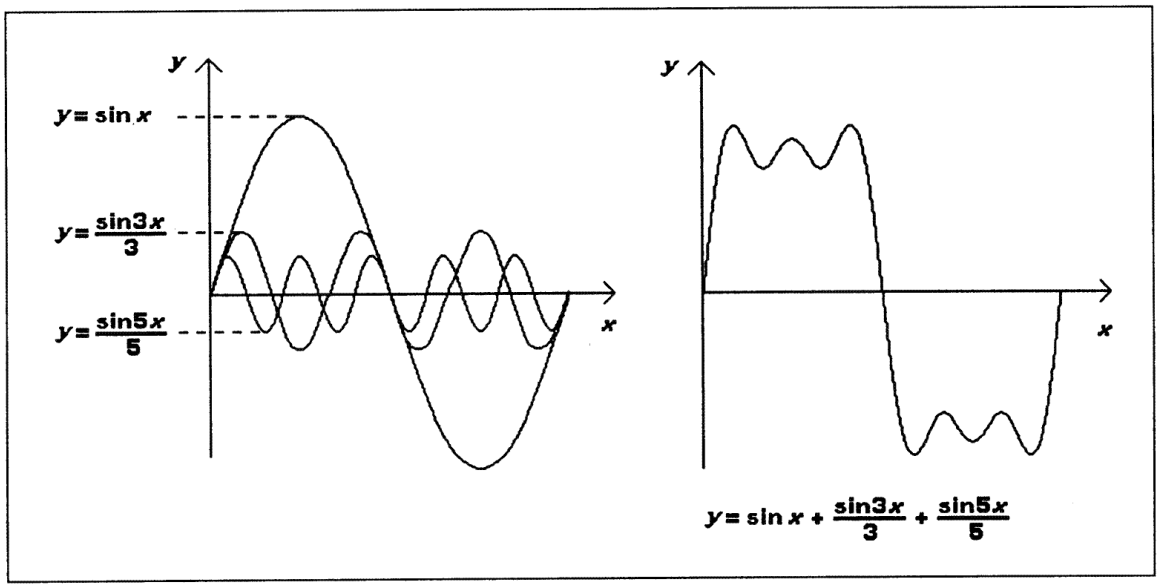


Abb. 1.5: Konstruktion einer Rechteckwelle mit 3 Sinusanteilen

$$f(x) = \frac{a_0}{2} + \sum_{k=1}^{\infty} a_k * \cos(k*x)$$

mit den Koeffizienten

$$a_k = \frac{1}{\pi} * \int_{-\pi}^{+\pi} (f(x) * \cos(k*x))\, dx \qquad (k = 0,1,2,...)$$

$$b_k = 0 \qquad \text{(für alle k)}$$

beziehungsweise

$$f(x) = \frac{a_0}{2} + \sum_{k=1}^{\infty} a_k * \sin(k*x)$$

mit

$$a_k = 0 \qquad \text{(für alle k)}$$

$$b_k = \frac{1}{\pi} * \int_{-\pi}^{+\pi} (f(x) * \sin(k*x))\, dx \qquad (k = 0,1,2,...)$$

Formel 6

Wie in Abb. 1.5 zu sehen, läßt sich durch Addition von nur drei Sinusfunktionen bereits eine zwar noch nicht sehr exakte, aber doch schon erkennbare Annäherung an die hier gewünschte Funktion, die Rechteckfunktion, erreichen. Diese und die anderen in der Abb. 1.6 dargestellten Wellenformen, Sinus-, Dreieck-, Rechteck- und Sägezahnform, sind relativ elementare mathematische Funktionen und daher sehr einfach mit elektronischen Mitteln zu erzeugen. Deshalb begegnet man ihnen auch überall in der elektronischen Musik. Hier wird erkennbar, wie Anzahl und Intensität der enthaltenen Oberwellen ansteigen, je weiter sich die Wellenform von der reinen Sinusfunktion entfernt.

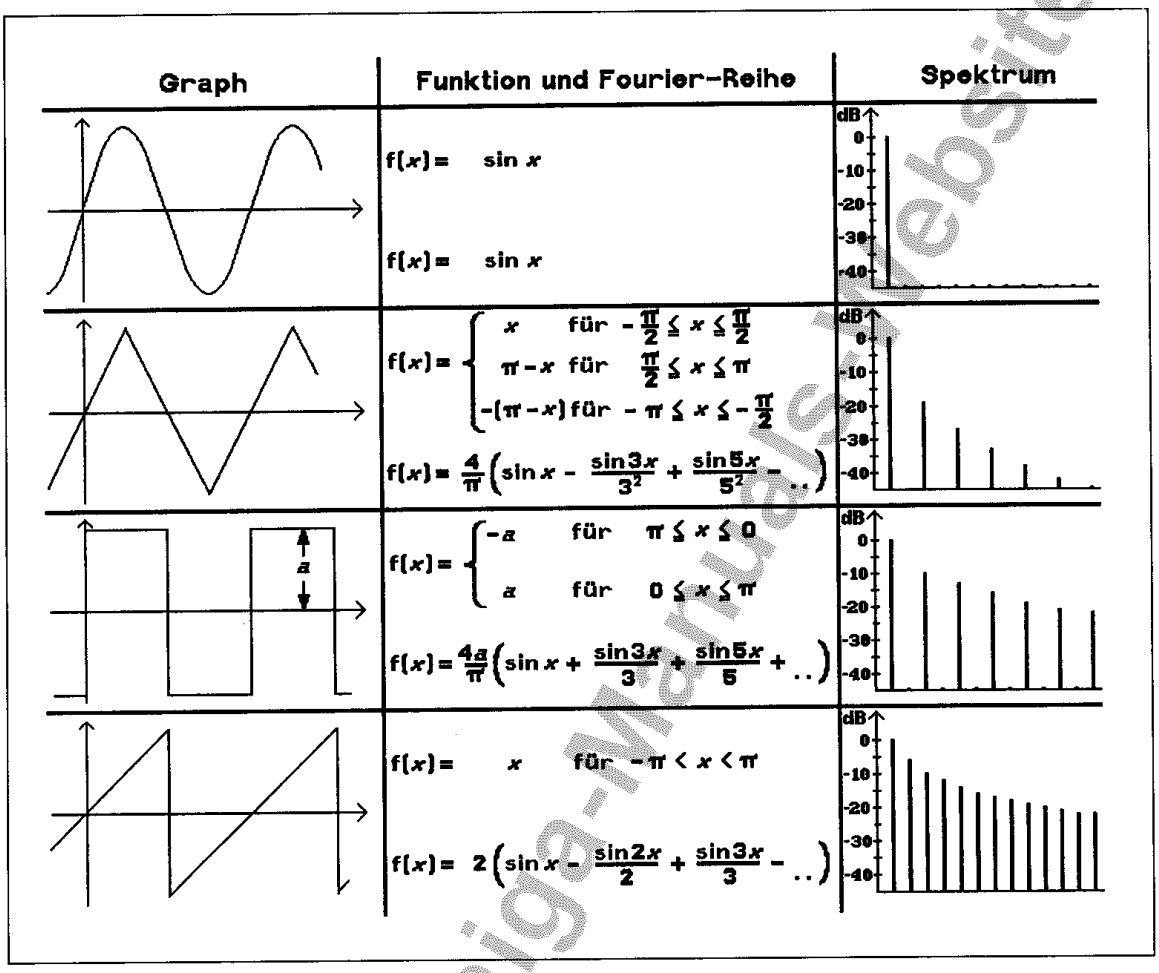


Abb. 1.6: Verschiedene Fourier-Entwicklungen

Periodizität und Rauschen

Die bisher besprochenen Typen von Schwingungen haben mit den Klängen von Musikinstrumenten eines gemeinsam: Sie bestehen alle aus einer zwar unendlichen, ober doch definierbaren Anzahl von Teilschwingungen. Weil sie sich auch allein aus Funktionen zusammensetzen lassen, die einen festen Frequenz- und Periodenabstand aufweisen, wie z.B. bei der Fourier-Analyse die ganzzahligen Vielfachen der Grundschwingung, sind sie periodisch und ihr Verlauf damit durchgehend vorhersagbar. Diese Verteilung von Frequenzkomponenten nennt man in der Physik ein diskretes Spektrum, da sich die einzelnen Anteile voneinander unterscheiden und trennen lassen.

Dies ist aber bei weitem nicht bei allen Schallereignissen der Fall, denen wir in der Natur begegnen. Da die meisten von ihnen nicht von physikalisch genau definierten Systemen mit bestimmten Eigenfrequenzen wie Luftsäulen, Saiten oder elektronischen Schwingkreisen erzeugt werden, sondern von Ereignissen eher zufälligen Charakters, sind sie nicht auf die oben beschriebene Weise darstellbar. Solche Schwingungen bezeichnet man im Audiobereich als Rauschen. Das Spektrum einer solchen Wellenform unterscheidet sich von denen der periodischen Funktionen

Taken from Amiga-Manuals-Website

vor allem dadurch, daß die einzelnen Frequenzanteile untereinander keinen bestimmten Abstand mehr haben, sondern unendlich nahe beieinanderliegen. Dies nennt man auch ein kontinuierliches Spektrum. Da der Frequenzbereich eines Rauschens im allgemeinen begrenzt ist, ist es auch hier sinnvoll, von einer Spektralverteilung zu sprechen und zu versuchen, diese zu ermitteln, was man mit Hilfe des sog. Fourier-Integrals erreichen kann. Ist die Verteilung entlang eines bestimmten betrachteten Frequenzbereiches genau ebenmäßig, d.h. alle Anteile in diesem Bereich weisen dieselbe Intensität auf, so spricht man auch von einem weißen Rauschen.

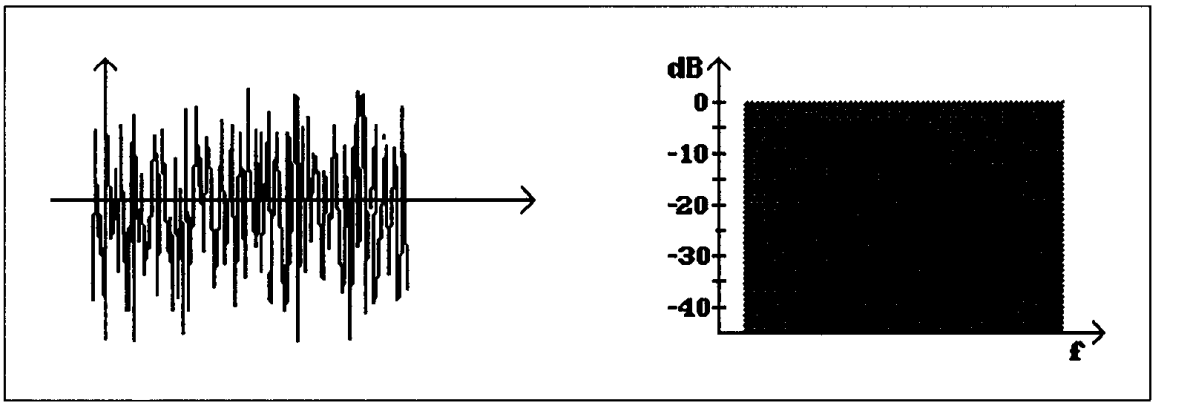


Abb. 1.7: Weißes Rauschen und Rauschspektrum

Das Rauschen ist die vorherrschende Wellenform in der Natur. Typische Beispiele für Rauschspektren sind das Windrauschen, bei dem die Zusammenstöße und Reibungen einer großen Anzahl von Luftteilchen zu einem vollkommen zufälligen Schalldruckverlauf führen, oder das thermische Rauschen eines elektronischen Bauteiles, das durch die Wärmebewegung der Moleküle der leitenden Schichten und die davon angeregte Bewegung der Elektronen verursacht wird.

1.1.3 Die Beschreibung akustischer Ereignisse

Im letzten Abschnitt wurde gezeigt, wie sich Schwingungsformen analog der Funktionsweise des menschlichen Gehörs analysieren und beschreiben lassen. Es wurden hierbei jedoch nur Funktionen betrachtet, die entweder periodisch sind oder, im Falle des Rauschens, zumindest ein gleichbleibendes Spektrum aufweisen und so für unser Gehör einen durchgehenden konstanten Klang darstellen. Die meisten Schallereignisse aber, denen wir in der Realität begegnen, ändern in ihrem Verlauf ihre Lautstärke, Tonhöhe und/oder spektrale Verteilung. In der Audiotechnik ist es daher notwendig, weitere Klassifikationskriterien für die Eigenschaften von akustischen Ereignissen aufzustellen.

Der spektrale Verlauf

Einen absolut beliebigen, nichtperiodischen Druckverlauf bezeichnet man als Geräusch. Im allgemeinen ist das Spektrum eines Geräusches nicht konstant, die Gewichtung der Frequenzanteile ändert sich also in seinem Verlauf. So ist es z.B. eine typische Eigenschaft des Windrauschens, bei starkem Wind heller zu klingen als bei schwacher Luftbewegung. Deshalb ist es

möglich, auf der Grundlage eines weißen Rauschens, das man durch einen Frequenzfilter leitet und die Filtercharakteristik ungefähr im Rhythmus von Windböen ändert, eine recht gute Nachahmung einer romantischen Sturmnacht zu erzielen. Genauso lassen sich periodische Wellenformen durch den Einsatz von Filtern in ihrer Spektralverteilung beliebig gewichten. In der menschlichen Sprache wird z.B. die Schwingung der Stimmbänder durch die Artikulationsorgane so verändert, daß bestimmte Teilschwingungen, Formanten genannt, besonders stark hervortreten und damit den Klang der verschiedenen Vokale A, E, I, O, U ausmachen. Dasselbe wird erreicht, wenn man eine sehr oberwellenreiche Wellenform aus einem Synthesizer, wie z.B. Rechteck oder Sägezahn, einer entsprechenden Filterbehandlung unterzieht. Hieraus wird erkennbar, wie sich komplexe, veränderliche Klangbilder mit Hilfe von wenigen Grundwellenformen und einer dynamischen Filterung beschreiben und nachahmen lassen.

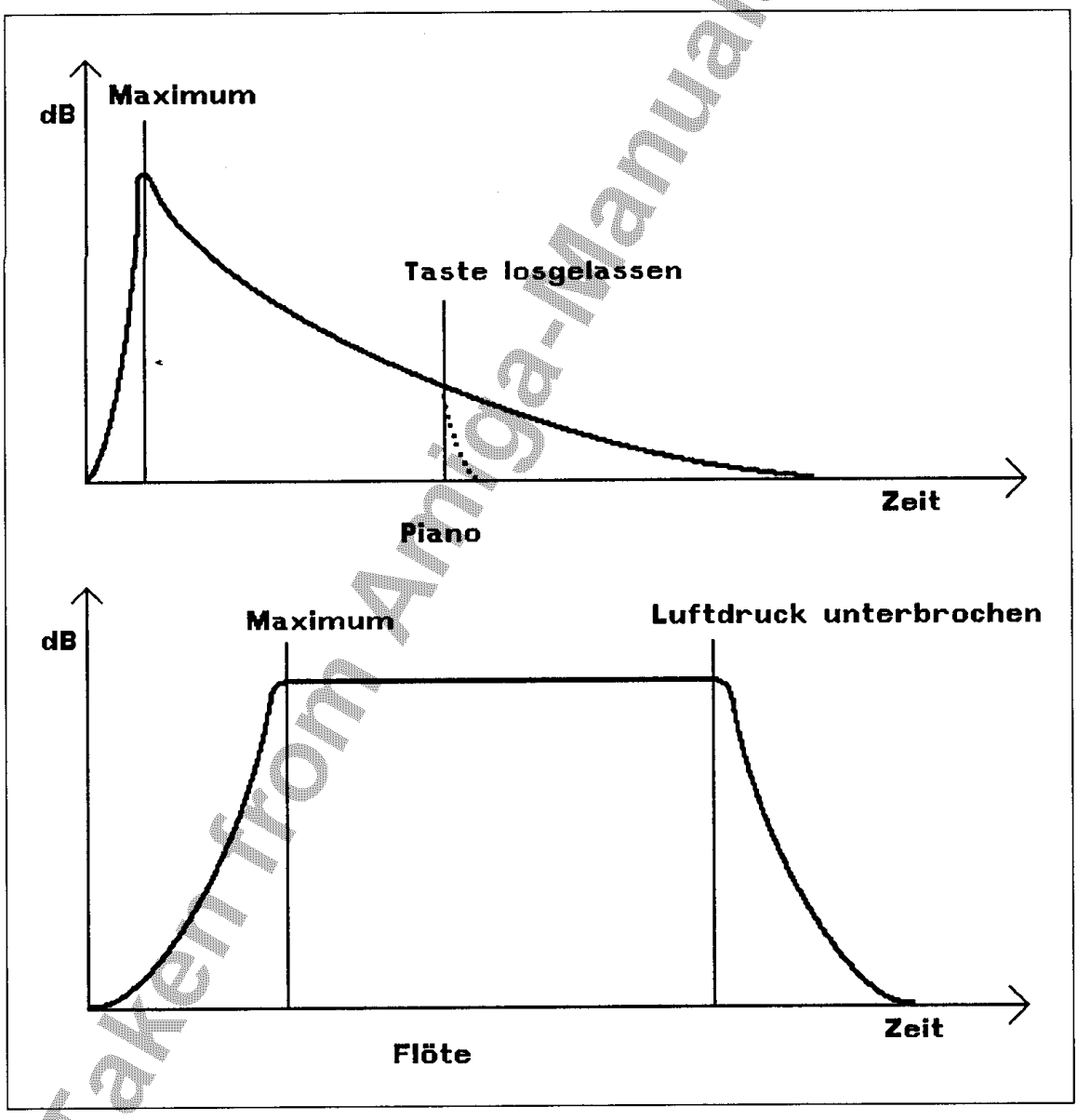


Abb. 1.8: Hüllkurve von Klavier und Flöte

Taken from Amiga-Manuals.Mobile

Hüllkurven

Eine fast noch gewichtigere Eigenschaft eines Schallereignisses als seine Spektralverteilung ist sein Lautstärkeverlauf. Eine Flöte beispielsweise klingt nicht nur deswegen weich und sanft, weil sie nur eine geringe Anzahl von Oberwellen aufweist, sondern auch wegen des sogenannten Anblasverhaltens, also der zeitlichen Verzögerung, die sich zwischen dem Beginn des Hineinblasens und dem vollen Erklingen des Tones ergibt. Im Gegensatz dazu hat das Anzupfen einer Gitarrensaite oder der Anschlag eines Klaviers einen sehr harten Charakter, da von der ersten Schwingungsphase an die volle Lautstärke vorhanden ist.

Diesen Verlauf der Gesamtlautstärke eines Tones bezeichnet man auch als seine Hüllkurve (engl.: envelope). Um nun die Hüllkurve eines beliebigen vorgegebenen Schallereignisses, vorzugsweise des Klanges eines Instruments, einigermaßen treffend beschreiben zu können, ohne dabei die Lautstärke jedes einzelnen Moments berücksichtigen und wiedergeben zu müssen (dies wäre eine ziemlich unhandliche Angelegenheit), teilt man in der Audiotechnik die Anschlag- und Ausklingphasen eines Tones in vier einzelne, in sich lineare Abschnitte ein: Attack, Decay, Sustain und Release.

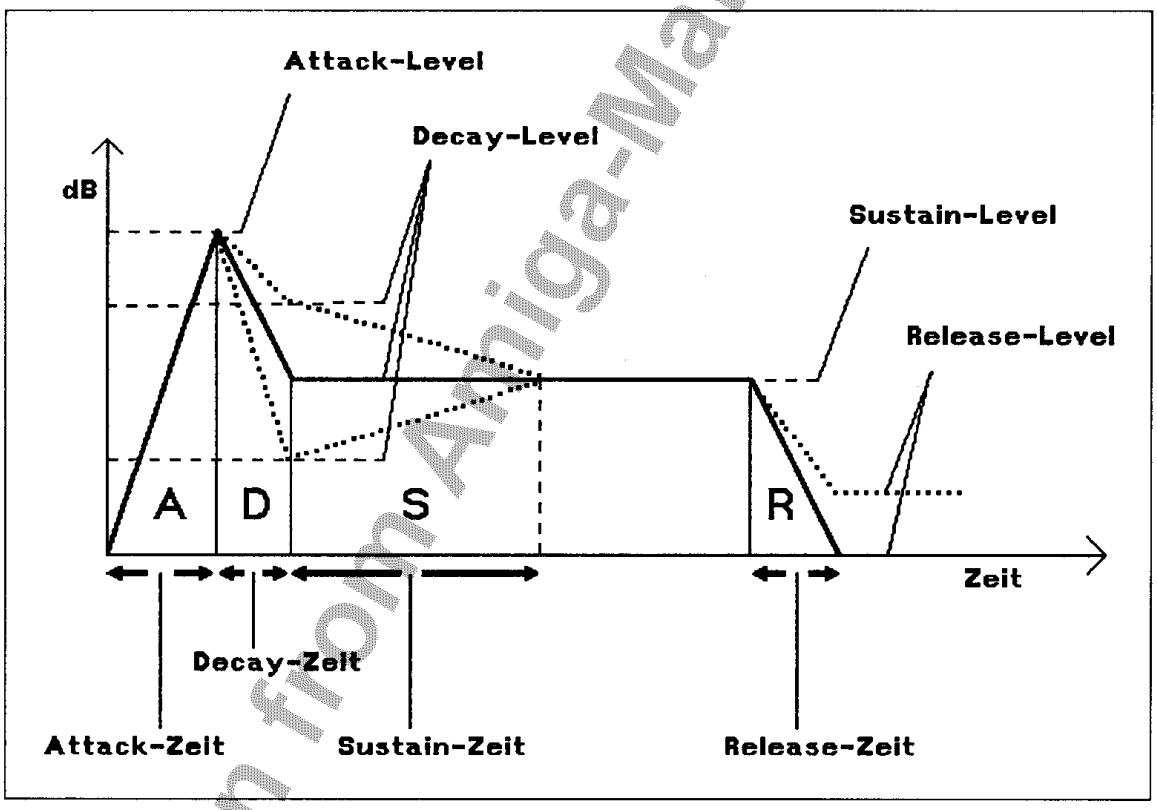


Abb. 1.9: ADSR-Hüllkurvenabschnitte

Für den Höreindruck sind die Zeitspannen, die von jeder Phase eingenommen werden, die am meisten entscheidende Komponente; aber auch die Pegel, auf denen die einzelnen Phasen beginnen und enden, sind hier nicht ohne Bedeutung. Deshalb definiert man ADSR-Werte oft

Taken from Amiga-Manuals-Website

auch als den Wert der Endlautstärke der jeweiligen Phase, was in diesem Fall der Maximalwert zwischen Attack- und Decayphase wäre, kombiniert mit der Geschwindigkeit des Amplitudenänderung, der sog. Anstiegsrate. Multipliziert man diese mit der Differenz der beiden angrenzenden Pegelwerte, so erhält man die effektive Zeit, die vom letzten Amplitudenwert bis zum Erreichen des neuen Wertes bei der jeweiligen Anstiegsrate vergeht. Dies erlaubt eine noch flexiblere Darstellung eines Lautstärkeverlaufs, da die Lautstärken der einzelnen Phasen dann in beliebigen Verhältnissen zueinander stehen können.

Als Attack bezeichnet man die Zeitspanne, die vom Auslösen des Tones bis zum Erreichen der maximalen Lautstärke vergeht. Wie oben erwähnt, bewirkt eine kurze Attack-Dauer einen scharfen explosiven Ton, während eine lange Attack-Zeit einen weichen, gedehnten Toncharakter zur Folge hat.

Der Decay-Wert beschreibt die Dauer des Absinkens der Lautstärke vom Attack-Lautstärkewert auf ein konstantes Niveau oder, wie bei Instrumenten ohne konstante Haltephase, auf den 0-Wert. Besitzt ein Instrument auch keine Dämpfungsvorrichtung, wie die Dämpfer eines Klaviers, sondern schwingt stets frei aus, wie z.B. eine Glocke, kann der Lautstärkeverlauf allein mit diesen beiden Elementen, Attack und Decay, wiedergegeben werden.

Das Sustain bestimmt den Pegel, auf dem sich die Lautstärke nach der Decay-Phase stabilisiert. Dieser wird solange aufrechterhalten, wie die entsprechende Taste gedrückt oder ein Blasinstrument mit Luft versorgt wird. Eine solche Haltephase besitzen nur Instrumente, die durchgehend mit Energie zum Schwingen versorgt werden, wie z.B. eine Orgel, eine Flöte, eine Geige oder ein elektronisches Instrument. Der Lautstärkepegel kann sich auch während der Sustain-Phase noch ändern, wenn das Sustain-Level durch die Anstiegsrate ergänzt wird, mit der Klang vom Ende der Decay-Phase mit dem dort vorliegenden Lautstärkewert bis zum Erreichen des Sustain-Levels anschwellen oder leiser werden soll. Auf diese Weise lassen sich auch Töne mit einem sehr unnatürlichen Lautstärkeverlauf, wie z.B. einem »Stottern«, erzeugen.

Der Release-Wert schließlich bezeichnet die Dauer des Abklingens eines Tones, nachdem eine Taste losgelassen oder der Luftstrom eines Blasinstrumentes unterbrochen wurde. Dies entspricht auch dem Abdämpfen einer Gitarrensaite oder eines Schlagzeugbeckens mit der Hand. Bei elektronischen Instrumenten kann man die Endlautstärke dieser Phase, die bei natürlichen Instrumenten immer 0 ist, auch auf einen noch hörbaren Wert setzen, so daß jeder Ton quasi unendlich lange »nachhallt«.

Wie oben erwähnt, gibt es in der Audiotechnik zwei Interpretationen für die ADSR-Einteilung einer Hüllkurve: Im einfacheren Fall betrachtet man die Werte von Attack, Decay und Release als Zeitspannen und Sustain als den Haltepegel, wobei die sonstigen Pegel entweder als das Maximum oder als 0 definiert sind. Mehr Möglichkeiten gewinnt man jedoch, wenn man jeder dieser Phasen sowohl eine Anstiegsrate als auch einen Endlautstärkewert zuordnet.

In Abb. 1.10 sind verschiedene Lautstärkeverläufe auf diese Weise dargestellt. Bei der Imitation von Musikinstrumenten oder Geräuschen mit relativ einfachen elektronischen Vorrichtungen, wie einem Sinus- oder Rauschgenerator, geht man am besten so vor: Zuerst bestimmt man die dem Klang zugrundeliegende Wellenform, wobei eventuelle Änderungen derselben außer acht gelassen werden müssen, sofern man nicht über entsprechende variable Filter verfügt. Dann versucht

Textprobe aus Musikmarktwissen

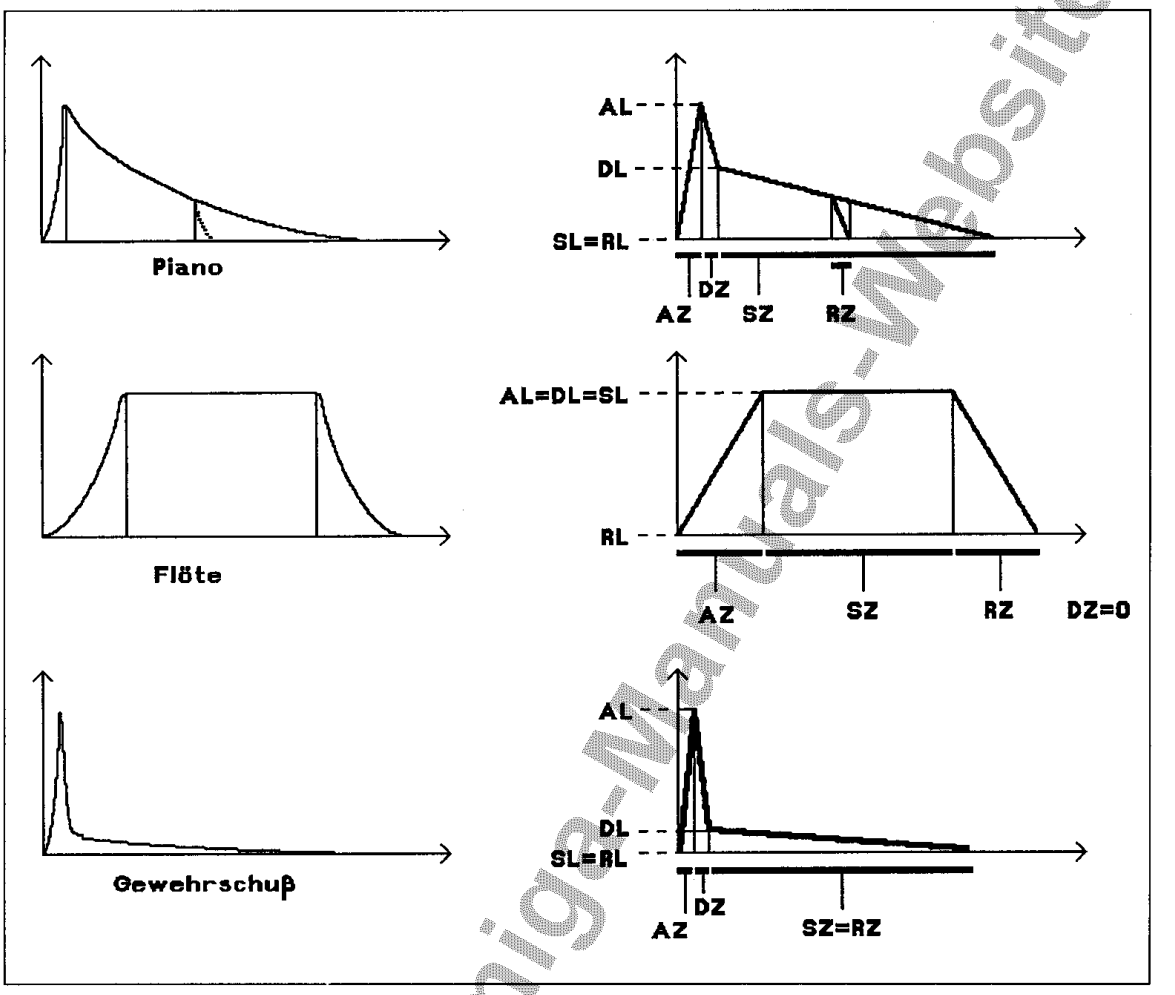


Abb. 1.10: Nachahmung der Hüllkurven verschiedener Instrumente

man, den geraden Verlauf der ADSR-Hüllkurve der vorliegenden Hüllkurve möglichst genau anzunähern, indem man sich an den jeweiligen Maximalwerten für Attack, Decay und Release orientiert und eventuelle Nichtlinearitäten vernachlässigt.

1.2 Die digitale Verarbeitung von Audiosignalen

Der Einsatz digitaler Verfahren hat sich in den letzten Jahren in so gut wie jedem Bereich der Audiotechnik mit großer Geschwindigkeit verbreitet und durchgesetzt. Sei es nun im HiFi-Bereich, wo die Hersteller von DAT-Recordern oder CD-Spielern Umsatzrekorde feiern, oder auf dem Sektor der Musikinstrument- und Studiotechnik, wo inzwischen digitale Synthesizer, Drummachines, Mischpulte und Aufnahmegeräte immer mehr die konventionellen analogen Geräte verdrängen, überall eröffnet die Digitaltechnik eine neue Dimension und setzt bisher nicht bekannte Maßstäbe hinsichtlich Qualität und Flexibilität der in der Audiotechnik eingesetzten Geräte. Wie am Beginn dieses Kapitels erwähnt, ergeben sich aus dem Verfahren, statt analoger Größen nur die Werte 0 und 1 zuzulassen, einige überraschende Eigenschaften der digitalen

Takes from Amiga-Manuals-Website

Bearbeitung von Audiosignalen, die in der konventionellen, analogen Weise nicht oder nur mit enormem Aufwand zu erzielen wären. Betrachtet man den Bereich der reinen Aufnahme- und Wiedergabetechnik, so fallen einem zuerst auf:

- Unempfindlichkeit von Übertragungen gegen Einstreuungen und bauteilspezifische Fehler wie Rauschen, Temperaturdrift etc.
- Keine Anhäufung von Fehlern und Verzerrungen bei mehrmaliger Aufnahme und Wiedergabe (Kopiereffekt)
- Praktisch kein Bandrauschen
- Absolut linearer Frequenzgang bis hinunter zu Gleichspannungen
- Keine Gleichlaufschwankungen
- Keine Beeinflussung der einzelnen Kanäle untereinander (Übersprechen)
- Perfekte Bandlöschung

Nimmt man noch den Bereich der Instrumentaltechnik hinzu, ergeben sich noch weitere Vorteile der digitalen Signalverarbeitung:

- Synthese von beliebig komplexen Wellenformen
- Keine Beschränkung hinsichtlich Wiedergabezeitpunkt oder Wiederholungen, sondern beliebiges Aneinanderreihen von Teilen vorliegender Daten möglich (Drum-Machine)
- Wiedergabe aufgenommener Klänge in beliebigen Tonhöhen durch Änderung der Ausgaberate (Sampler)

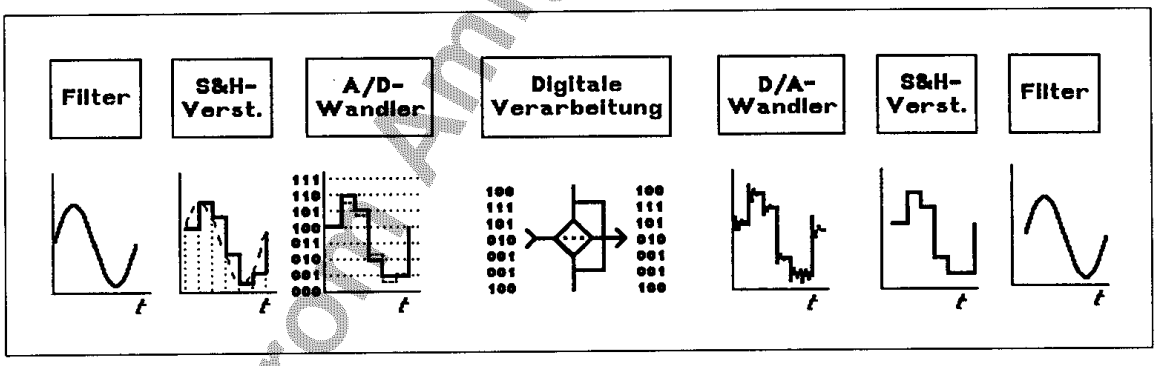


Abb. 1.11: Komponentenkette zur digitalen Signalverarbeitung

Es sind jedoch auch hier einige für die Digitaltechnik spezifische Probleme und Fehlerquellen zu beachten. In diesem Abschnitt sollen zunächst die Methoden zur Umsetzung von analogen Spannungen in digitale Werte betrachtet werden. Vor allem die durch die Abstufung und Rundung (Quantisierung) von Zeit- und Spannungswerten hervorgerufenen Verzerrungen sind hier von besonderem Interesse. Anschließend werden die Aspekte der Rekonstruktion, also der Digital/Analog-Umsetzung und Filterung, eingehend untersucht.

Taken from Amiga-Manuals-Website

1.2.1 Abtastung

Sample&Hold-Verstärker und A/D-Wandler
Das Kernstück desjenigen Teiles der in Abb. 1.11 dargestellten Komponentenkette, der für die Analog- nach Digital-Umsetzung zuständig ist, ist der A/D-Wandler. Er hat die Aufgabe, eine anliegende analoge Spannung auf einen bestimmten Befehl hin zu prüfen und in einen ganzzahligen numerischen Wert umzuwandeln. Dieser Befehl zur Probenentnahme (Sampling) wird in einem festen zeitlichen Abstand (Sampling-Rate) wiederholt, so daß man eine Folge von Zahlen erhält, die den Verlauf der anliegenden Spannung beschreiben.

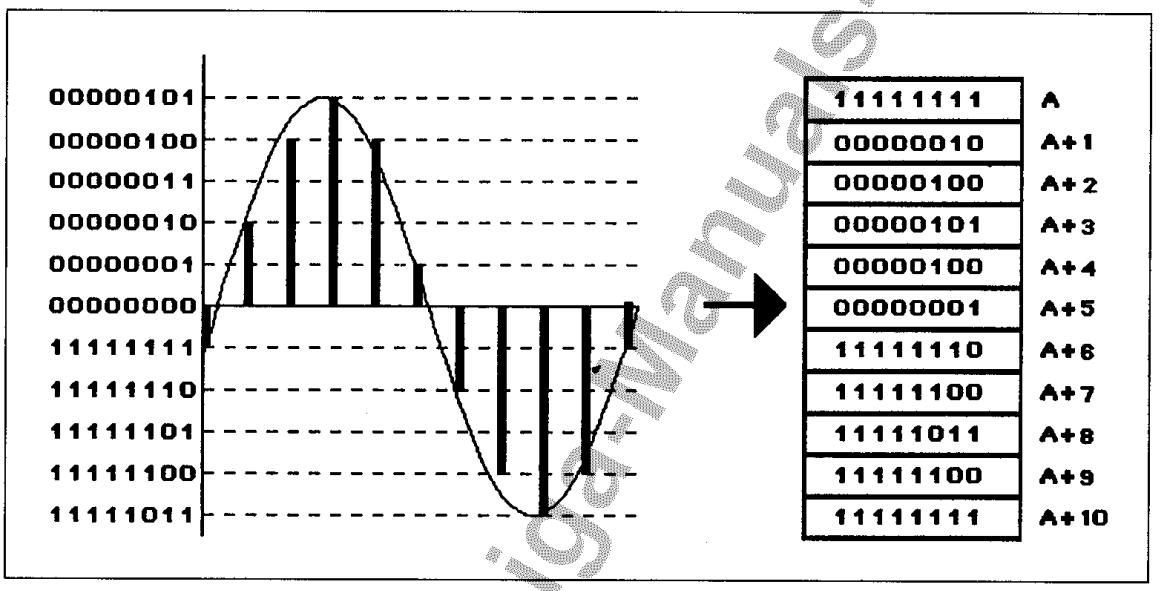


Abb. 1.12: Analog-Digital-Umsetzung

Binärzahlen

Zur Repräsentation von Zahlenwerten verwendet die gesamte Digitaltechnik eine Darstellung durch binäre Werte, auch Binary Digits oder kurz »Bits« genannt. Diese wurden gewählt, da sie nur die Werte 0 und 1 annehmen können, um so den Aufbau von digitalen Rechenwerken möglichst zu vereinfachen. Möchte man nun höhere Werte als 0 und 1 darstellen, so muß man diese binären Stellen in Form eines Stellenwertsystems, in diesem Fall des Dualsystems, anordnen. Im Dualsystem, ebenso wie im sonst gebräuchlichen Dezimalsystem, werden den einzelnen Bits bzw. den einzelnen Stellen die Basis des Systems, 2 bzw. 10, zur Potenz ihrer Stellennummer erhoben zugeordnet. Im Dualsystem heißt das, daß die Stelle mit dem kleinsten Wert (LSB = least significant bit) wie im Dezimalsystem mit dem Faktor 1 belegt wird, die nächsthöhere Stelle aber nicht zehnmal, sondern nur doppelt so hoch bewertet wird. Während also die Ziffern einer Dezimalzahl mit 1, 10, 100, 1000 oder in Potenzschreibweise 10^0, 10^1, 10^2, 10^3 etc. multipliziert werden müssen, sind die Bits einer Binärzahl mit 1, 2, 4, 8 oder 2^0, 2^1, 2^2, 2^3 etc. zu multiplizieren, um deren Wert zu erhalten. Der Wertebereich, den man mit einer Anzahl von n Bit darstellen kann, reicht daher von 0 bis 2^n-1, wobei man das höchstwertige Bit als MSB (Most Significant Bit)

Taken from Amiga-Manuals-Website

bezeichnet. Da man bei der Numerierung der Bits einer Binärzahl stets bei 0 beginnt (das niederwertigste Bit hat immer den Wert $2^0 = 1$), steht das MSB immer an der Stelle n–1, sein Wert beträgt also 2^{n-1}.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0		
128	64	32	16	8	4	2	1		
0	0	0	0	0	0	0	0	=	0
0	0	0	0	0	0	0	1	=	1
0	0	0	0	0	0	1	0	=	2
0	0	0	0	0	0	1	1	=	3
0	0	0	0	0	1	0	0	=	4
0	0	0	0	0	1	0	1	=	5
				•	•	•	•		

Abb. 1.13: Dualsystem

Das Zweierkomplement

Mit Hilfe des Dualsystems lassen sich also aus Bits positive ganze Zahlen konstruieren. Wie sieht es aber mit negativen Zahlen aus? Das Dualsystem liefert hierauf keine Antwort, da in ihm keine negativen Werte definiert sind. Es muß also eine Modifikation dieser Darstellungsform gefunden werden, die den Anwendungsbereich der Binärzahlen dementsprechend erweitert. Auch sollte die Durchführung elementarer mathematischer Operationen keine Schwierigkeit bedeuten. Eine Darstellung von negativen Binärzahlen muß also zusätzlich noch den Eigenschaften der binären Addition und Subtraktion entsprechen, d.h., jede Zahl muß von sich selbst subtrahiert oder mit der negativen addiert 0 ergeben. Darüber hinaus sollten auch einfache, schnelle Operationen wie Multiplikation durch Bitverschiebung etc. noch ihre Gültigkeit behalten.

Taken from Amiga-Manuals-Website

Eine Möglichkeit, dies zu realisieren, bildet die Zweierkomplement-Darstellung. Sie ist abgeleitet vom einfachen Signedbinary-Verfahren, bei dem das höchstwertige Bit einer Dualzahl als Vorzeichen und die restlichen Bits als der Absolutwert betrachtet werden, wobei im Vorzeichenbit traditionsgemäß 0 für einen positiven und 1 für einen negativen Wert steht. Beim Zweierkomplement ist dies im positiven Bereich zwar ebenso, jedoch unterscheidet es sich von der Signed-binary-Methode bei der Darstellung von negativen Zahlen. Um die zu einer Zahl gehörige negative Zahl zu erhalten, bildet man zunächst deren binäres Komplement, d.h., man setzt für jedes 1-Bit eine 0 ein und umgekehrt, jedes 0-Bit auf 1. Dann folgt das Entscheidende an der Zweierkomplement-Methode: Zu dem so erhaltenen Wert addiert man noch 1 hinzu, um eine korrekte binäre Addition und Subtraktion zu ermöglichen. Daß dies nötig ist, sieht man an der Tatsache, daß ohne diese Korrektur in der Darstellung keine zusammenhängende Abfolge der Zahlenwerte mehr gewährleistet wäre, da für die Stelle 0 zwei Werte vorhanden wären, nämlich eine positive und eine negative 0. Um diese Unstimmigkeit mit der mathematischen Realität zu beseitigen und die Programmierung zu vereinfachen, führte man das Zweierkomplement fast überall in der Digitaltechnik als gebräuchliche Darstellung von Integer- (ganzen) Zahlen ein.

In der folgenden Tabelle sind die Umrechnungs-Werte für eine 8 Bit breite Zweierkomplement-Darstellung aufgeführt:

dezimal	Zweierkomplement	dezimal	Zweierkomplement
0	00000000		
1	00000001	–1	11111111
2	00000010	–2	11111110
3	00000011	–3	11111101
4	00000100	–4	11111100
5	00000101	–5	11111011
6	00000110	–6	11111010
7	00000111	–7	11111001
8	00001000	–8	11111000
9	00001001	–9	11110111
10	00001010	–10	11110110
11	00001011	–11	11110101
...			
126	01111110	–127	10000001
127	01111111	–128	10000000

Der A/D-Wandler

Im Bereich der Analog/Digital-Umsetzung haben sich verschiedene Methoden etabliert. Viele A/D-Wandler verwenden eine Darstellung ihres Wertebereiches von 0 bis 2^n-1, wobei der »Nullpunkt« bei 2^{n-1} liegt, andere liefern ihre Ausgabewerte in der Zweierkomplement-Form. Da dies für den Aufbau des Wandlers meist keinen wesentlichen Unterschied bedeutet, existieren auch viele Versionen mit umschaltbarer Darstellungsart.

Taken from Amiga-Manager Note

Grundsätzlich unterscheidet man die folgenden Typen von A/D- Wandlern:

❑ Parallelwandler: Bei dieser Art von A/D-Wandlern existiert für jeden einzelnen Wert, der unterschieden werden soll, eine eigene Komparatorstufe, d.h. eine Vergleichseinheit, die entscheidet, ob der anliegende Spannungswert größer oder kleiner als der jeweilige Vergleichswert ist. Da diese Sorte eine sehr kurze Öffnungszeit aufweist, d.h. eine geringe Zeitspanne vom Start des Wandlungsvorganges bis zum Anliegen gültiger Daten, werden diese Wandler hauptsächlich in zeitkritischen Bereichen wie z.B der Videotechnik eingesetzt. Da aber bei einem Wertebereich von n Bit eine Anzahl von 2^n Komparatoren notwendig ist, was z.B. bei den beim CD-Player verwendeten 16 Bit eine Menge von 65536 Stück wäre, steigen die Kosten mit höherer Auflösung überproportional an, wodurch ihre Verwendung im Audiobereich unrentabel wird.

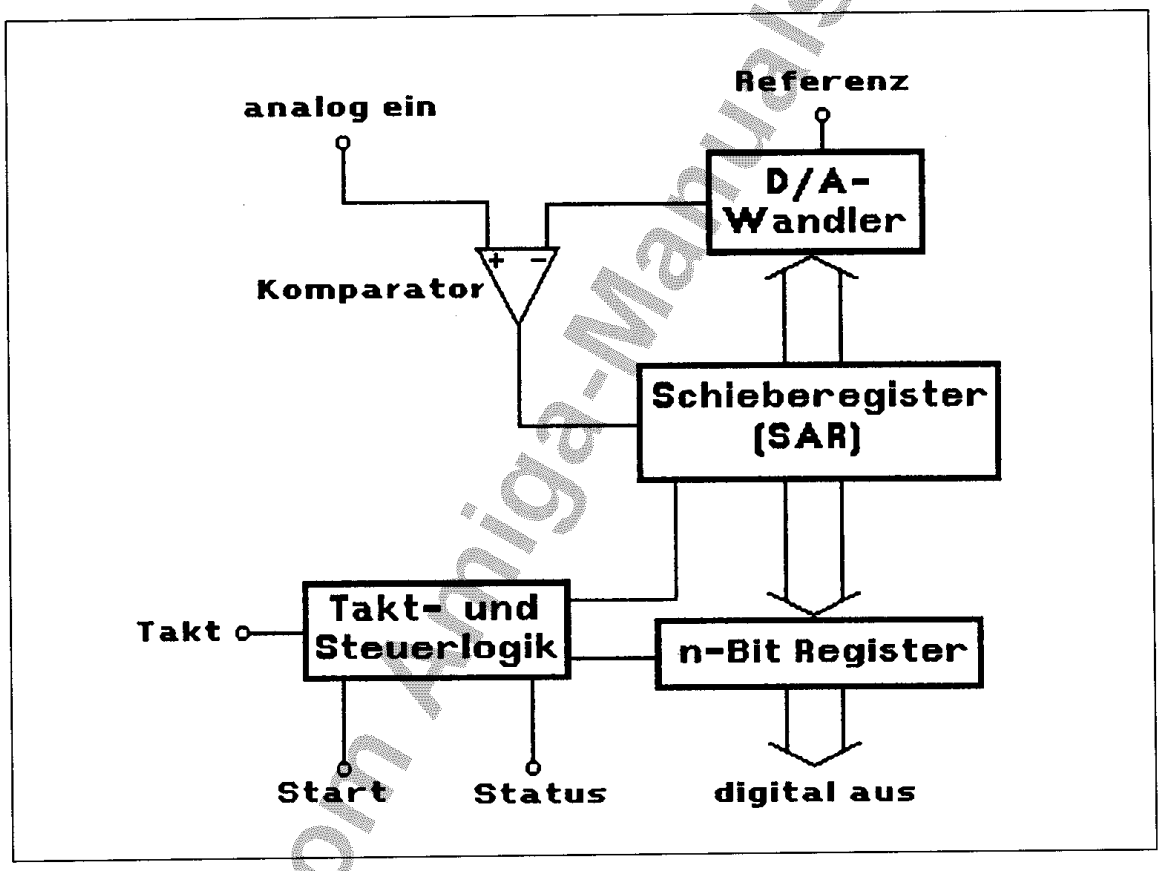


Abb. 1.14: A/D-Wandler mit schrittweiser Annäherung

❑ Wandler mit schrittweiser Annäherung (engl.: successive approximation): Solche Wandler errechnen ihr Ergebnis, indem sie die anliegende Spannung nacheinander mit Spannungswerten vergleichen, die im Verhältnis zur Maximalspannung (Referenzspannung) genau den Stellenwerten der einzelnen Bits entsprechen. Zuerst wird entschieden, ob die anliegende Spannung größer oder kleiner ist als die Hälfte der Referenzspannung und das höchstwertige Bit danach

Taken from Amiga-Manuals Website

gesetzt, dann wird ein Viertel der Referenzspannung hinzugefügt, wieder verglichen und dementsprechend das nächstuntere Bit gesetzt, dann dasselbe mit einem Achtel der Referenzspannung und so fort (siehe Abb. 1.15). Hierbei wird nur eine Vergleichseinheit benötigt, was gegenüber dem letztgenannten Wandlertyp einen hohen Preisvorteil bietet. Da der Vorgang der schrittweisen Spannungsannäherung jedoch mehrere Vergleichsphasen braucht, um zum endgültigen Ergebnis zu gelangen, vervielfacht sich die Wandlerzeit im Vergleich zu den Parallelwandlern ungefähr um die Anzahl der darzustellenden Bits, was bereits im Audiobereich zu der Frage führt, ob ein Wandler dieser Sorte für die hier herrschenden Ansprüche geeignet ist.

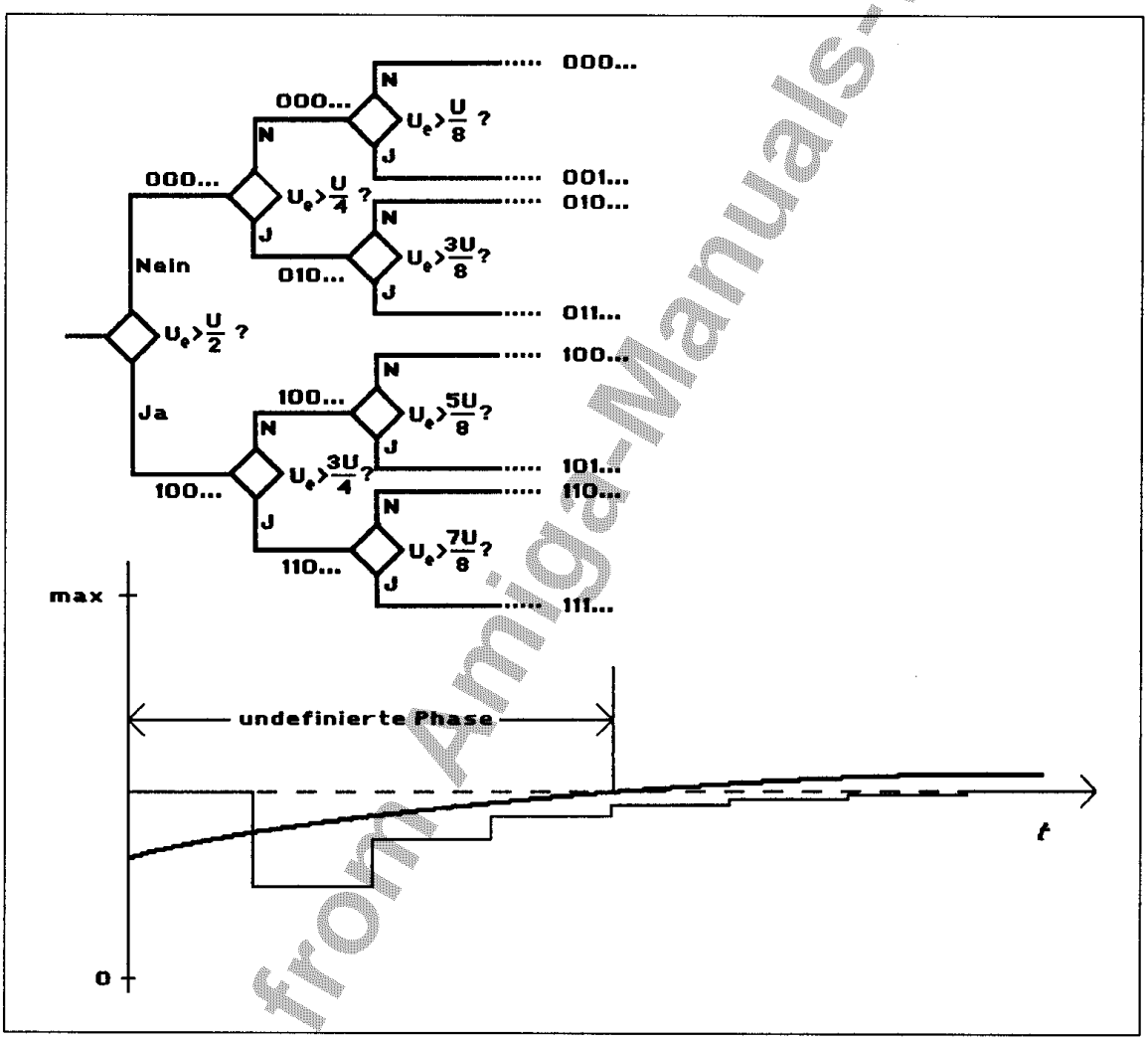


Abb. 1.15: Entscheidungskette bei der schrittweisen Annäherung

Betrachtet man nämlich die Phase vom Beginn der Wandlung bis zur Beendigung des letzten Vergleichsvorganges, so fällt auf, daß sich die anliegende Spannung inzwischen auch soweit verändert haben könnte, daß der vom Wandler ausgegebene Wert nicht mehr dem zum Zeitpunkt

Taken from Amiga-Manuals-Website

des Wandlungsbeginns anliegenden Spannungswert entspricht. Das Ergebnis wäre eine Fehlabtastung besonders von hohen Frequenzen, da bei diesen die Spannungspegel eine besonders hohe Änderungsgeschwindigkeit aufweisen. Sehen wir uns das an einem Beispiel an: Die meisten A/D-Wandler mit sukzessiver Approximation der mittleren Preisklasse haben Öffnungzeiten um 10 µs, können also maximal um die 100000 Wandlungen pro Sekunde ausführen. Nimmt man eine noch gut hörbare Sinusschwingung, beispielsweise mit einer Frequenz von 10 Kilohertz und einer Amplitude von 1 Volt, so stellt man fest, daß sich während der Öffnungszeit der Spannungspegel im ungünstigsten Fall bereits um 0.6 Volt, also um 60% geändert hat! Dies hat natürlich eine große Fehlerkomponente in den abgetasteten Werten zur Folge, die sich in Form von Verzerrungen bemerkbar macht. Allerdings zeigt die Praxis, daß trotz der theoretisch ermittelten Grenzfrequenz, bis zu der ein Wandler in Näherung exakt arbeitet, noch bei viel höheren Frequenzen brauchbare Ergebnisse erzielt werden.

Allgemein lassen sich der Amplitudenfehler und die daraus resultierenden Anforderungen an die Eingangsfrequenzen bzw. die Öffnungszeit des Wandlers so formulieren: Da eine Sinusschwingung an ihrer »steilsten« Stelle, der 0, genau eine Steigung von 1 aufweist, ist die Änderung des Spannungspegels, dU, beschreibbar durch

$$\Delta U = U_{max} * t_w * 2\pi * f$$

Formel 7

wobei U(max) die maximale Spannung, t(w) die Öffnungszeit des Wandlers und f die abgetastete Frequenz ist. Betrachtet man nun das Verhältnis zwischen diesem Amplitudenfehler und dem Maximalwert der Eingangsspannung

$$\frac{\Delta U}{U_{max}} = t_w * 2\pi * f$$

Formel 8

so ergibt sich für einen maximal tolerierbaren Fehler von +– 1/2 LSB, also

$$\frac{\Delta U}{U_{max}} = 2^{-(n-1)}$$

Formel 9

bei gegebener Frequenz eine zulässige Öffnungszeit des Wandlers von

$$t_w = \frac{1}{2^{-n} * \pi * f}$$

Formel 10

und bei gegebener Öffnungszeit eine höchstmögliche Frequenz von

$$f_{max} = \frac{1}{2^{-n} * \pi * t_w}$$

Formel 11

Der S&H-Verstärker

Eine Lösung für dieses Problem stellt der sogenannte Sample&Hold-Verstärker dar. Dieser hat die Aufgabe, eine am Eingang anliegende Spannung von einem bestimmten Moment an konstant-zuhalten, damit ein am Ausgang angeschlossener A/D-Wandler die Spannung in Ruhe analysieren kann. Im wesentlichen besteht eine solche S&H-Einheit lediglich aus einem Kondensator und einem Schalter, von dem der Kondensator jeweils für kurze Zeit mit der Eingangsspannung verbunden wird, um den Wert der Spannung anzunehmen. Dann wird die Verbindung zum Eingang unterbrochen, so daß bis zum Beginn des nächsten Wandlungzyklus am Kondensator eine unveränderliche, konstante Spannung greifbar ist.

Ein S&H-Verstärker bewirkt also die Umwandlung einer Schwingung in eine Treppenfunktion, wobei die einzelnen Stufen den zeitlichen Abstand der einzelen Abtastbefehle haben, allerdings noch keine festen Spannungsabstände aufweisen. Die Auswirkungen, die diese Umwandlung in eine Treppenfunktion auf das Frequenzspektrum hat, sind Gegenstand des Abschnitts 1.2.1.3. Die Effekte der Überführung von analogen Spannungswerten in ganzzahlige Werte und die sich daraus ergebenden Rundungsfehler werden im nächsten Abschnitt besprochen.

Die Auflösung des A/D-Wandlers

Ausgehend von der Tatsache, daß das Eingangsignal durch einen S&H-Verstärker lange genug konstantgehalten wird, um korrekt analysiert zu werden, wenden wir uns nun der eigentlichen Aufgabe des A/D-Wandlers zu, der Umwandlung der Spannungswerte in ganze Zahlen. Neben der Häufigkeit, mit der diese Abtast- und Umsetzvorgänge ausgelöst werden (Sampling-Rate), ist die Anzahl der verschiedenen Werte, in die der zulässige Bereich des Wandlers unterteilt ist (Auflösung), wohl die wichtigste Eigenschaft einer Analog/Digital-Umsetzung im Hinblick auf

ihre Qualität und Wiedergabetreue. Wird nämlich ein analoger Spannungswert auf einen digitalen, ganzahligen Wert gerundet oder der Nachkommateil einfach abgeschnitten, so kann der hierbei entstehende Fehler nicht größer sein als eine dieser »Quantisierungsstufen«, d.h. als der Abstand, der die einzelnen digitalen Werte voneinander trennt. Da die Menge der mit einer Dualzahl darstellbaren Werte von der Anzahl ihrer Stellen abhängig ist, leitet sich also die Umsetzungsqualität eines A/D-Wandlers im wesentlichen von der Anzahl der Bits ab, die dieser liefert.

Um die Effekte der Amplitudenquantisierung von denen der zeitlichen Abtastung besser unterscheiden zu können, wird im folgenden von der Tatsache der zyklischen Probenentnahme abgesehen und eine in dieser Hinsicht »ideale«, also pro Zeiteinheit unendlich oft durchgeführte Abtastung angenommen. Abb. 1.16 zeigt eine unter solchen Bedingungen digitalisierte Wellenform. Wie im letzten Abschnitt erläutert, läßt eine Anzahl von n Bit die Unterteilung des Spannungsbereichs in 2^n Werte zu, das kleinste unterscheidbare Spannungsintervall wird also durch jedes hinzugenommene Bit halbiert. Dementsprechend verkleinert sich natürlich auch der maximale Fehler, also die Differenz zwischen dem tatsächlich anliegenden Spannungswert und seiner vom A/D-Wandler wiedergegebenen digitalen Entsprechung. Diese durch die Anzahl der in der Darstellungsbreite verwendeten Bits bedingte Genauigkeit wird auch als die Auflösung eines A/D-Wandlers bezeichnet.

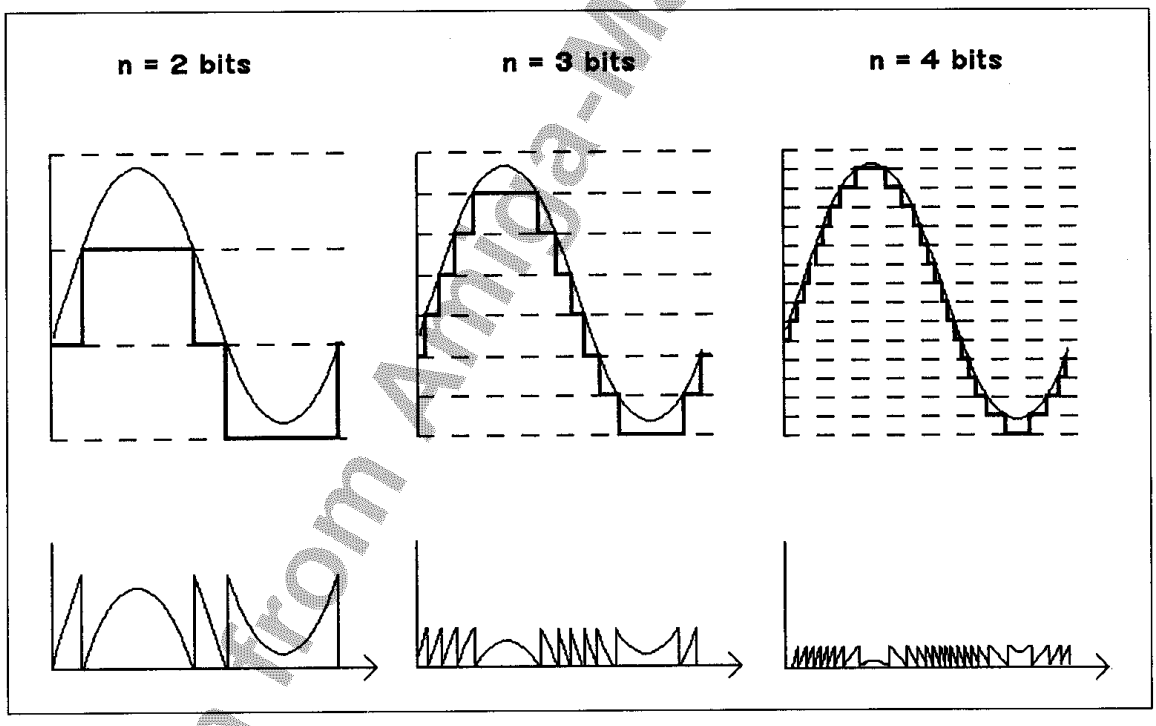


Abb. 1.16: Quantisierungsfehler bei verschiedenen Auflösungen

Der Wandler kann sich, wie erwähnt, bei der Anpassung der analogen Spannungen an die diskreten Zahlenwerte zweier Methoden bedienen: entweder einer Rundung im mathematischen Sinne, bei der unterhalb der Mitte zwischen zwei Werten ab- und darüber aufgerundet wird, oder

Taken from Amiga-Manuals-Website

der einfachen Vernachlässigung des Nachkommaanteils, wie in Abb. 1.16 dargestellt, wobei nur abgerundet wird. Verwendet der Wandler jedoch nur positive Werte oder die Zweierkomplement-Darstellung, so entspricht die mathematische Rundung genau der einfachen Abrundung des um ein halbes Rundungsintervall erhöhten Wertes, die sich aus den beiden Verfahren ergebenden Gesamtfehler haben also die gleiche statistische Verteilung. Dies bezeichnet man auch als die Symmetrie des Rundungsfehlers. Aufgrund dieser Tatsache können wir die Unterschiede zwischen den beiden Methoden vernachlässigen und zur Vereinfachung nur die Abrundung berücksichtigen.

Störabstand und Quantisierungsrauschen

Der Quantisierungsfehler bei einer A/D-Wandlung ist im allgemeinen in seinem Verlauf unvorhersagbar und ist deshalb einem Rauschen ähnlich, weshalb man es auch als »Quantisierungsrauschen« bezeichnet. Da eine Quantisierungsgstufe dem Wert des niederwertigsten Bits der Darstellung entspricht

$$1\ \mathrm{LSB} \mathrel{\widehat{=}} \frac{U_{max}}{2^n}$$

Formel 12

und da der Fehler maximal eine Quantisierungstufe betragen kann, spricht man in diesem Fall von einem Quantisierungsrauschen mit der Amplitude 1 LSB. Eine der wesentlichsten Eigenschaften eines Elementes in der Audiotechnik, der Störabstand, d.h. das Verhältnis zwischen der maximalen Lautstärke und der Lautstärke der Störgeräusche, läßt sich bei einem A/D-Wandler also direkt durch die Anzahl der Bits seiner Darstellung ermitteln. Es läßt sich daher schreiben:

$$\text{Rauschabstand in dB} = 20 * \log(2^n)$$

wobei log - dekadischer Logarithmus
n - Anzahl der Bits

Formel 13

Mit jedem Bit, das man bei einer Darstellung berücksichtigt, halbiert sich mit den Quantisierungsintervallen der zu erwartende Fehler und verdoppelt sich also der Rauschabstand. Da ein Verhältnis von 1 : 2 einen dB-Wert von

$$\text{dB-Wert (2)} = 20 * \log 2 \approx 6$$

Formel 14

hat, bedeutet ein zusätzliches Bit eine Rauschabstandverbesserung bzw. einen Dynamikgewinn von ungefähr 6 dB. Dieser läßt sich also auch so abschätzen:

$$\text{Rauschabstand in dB} = n * 6$$

wobei n - Anzahl der Bits

Formel 15

Eine Auflösung von 8 Bit, wie beim Amiga üblich, ermöglicht also einen Rauschabstand von ungefähr 48 dB. Die von einem CD-Player verwendeten 16 Bit erzielen sogar 96 dB.

Die Aussteuerung bei der A/D-Umsetzung

Bei den vorangegangenen Überlegungen bezüglich des Störabstandes wurde jedoch stets davon ausgegangen, daß die abgetastete Schwingung genau die maximal mögliche Amplitude hat, den durch die Auflösung vorgegebenen Wertebereich also voll ausnutzt. Da dies bei den real in der Audiotechnik vorkommenden Wellenformen selten der Fall ist, muß man sich die Frage stellen, welche Auswirkungen eine Unter- und Überschreitung dieses optimalen Aussteuerungspegels auf den Störabstand und damit auf die Klangqualität haben kann. Beschäftigen wir uns zuerst mit dem Fall der Unteraussteuerung, also der unzureichenden Abdeckung des zur Verfügung stehenden Wertebereichs. Wie in Abb. 1.17 zu sehen, treten in der Praxis oft Lautstärkeverläufe mit einer langen Ausklingphase auf, wie z.B. bei einem Klavier. Hier ist folgender Effekt zu beobachten: Je geringer die Lautstärke des ausklingenden Tones wird, desto kleiner wird auch das Verhältnis zwischen ihr und den Quantisierungsintervallen, desto ungünstiger also der Signal-/Rauschabstand. Diese Verminderung fällt bei mittleren Lautstärken noch nicht sonderlich ins Gewicht, doch spätestens, wenn der Wandler nur zwischen wenigen oder im Extremfall nur noch zwischen zwei Werten hin- und herspringt, wird der Klangcharakter so verstümmelt, daß der Unterschied auch bei sehr geringen Lautstärken störend auffällt. Das führt dazu, daß sich die Tonqualität beim Ausklingen eines Tones, obwohl die durch die Quantisierung erzeugte Rauschleistung stets gleich bleibt, subjektiv gesehen drastisch verschlechtert.

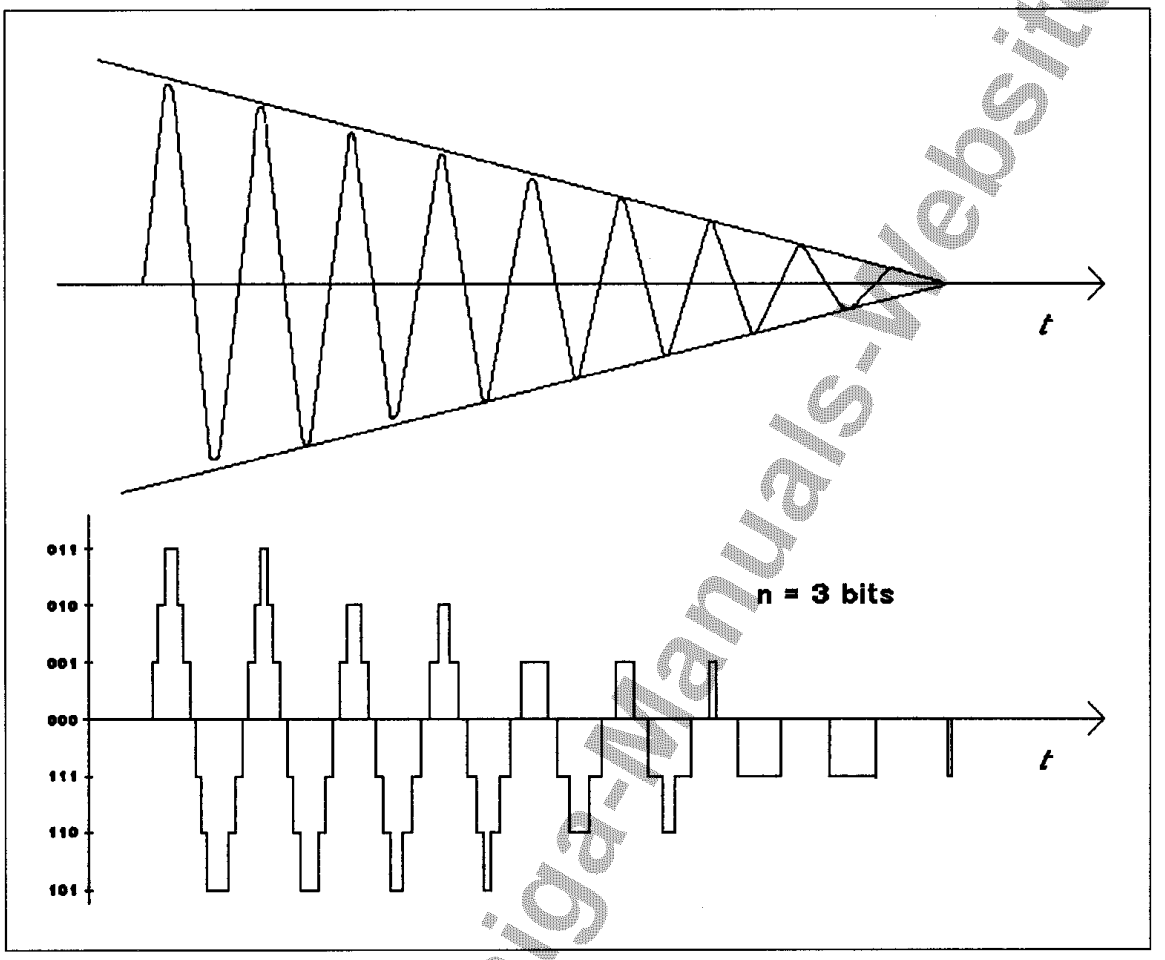


Abb. 1.17: Unteraussteuerung und Granularrauschen

Diesen Effekt bezeichnet man auch als Granularrauschen. Im Bereich der digitalen Aufnahme- und Wiedergabesysteme hat dieser eine große Bedeutung und schafft ziemliche Probleme, weshalb hier auch spezielle Verfahren zur Reduktion seiner Hörbarkeit eingesetzt werden, z. B. indem man es durch ein zusätzliches Rauschen von ungefähr 1 LSB Amplitude zu »maskieren« versucht. Auf dem Sektor der digitalen Musikinstrumente, also bei den Samplern, ist es im allgemeinen möglich, diesen unangenehmen Effekt durch die Einhaltung einer bestimmten Mindestaussteuerung weitgehend zu vermeiden. Um geringe Lautstärken zu erzeugen, sollte hier nicht die digitale Aussteuerung verwendet werden, sondern möglichst eine den vollen Wertebereich verwendende digitale Wellenform in Verbindung mit einem analogen Lautstärkeregler, damit auch bei kleiner Lautstärke der maximal durch die Auflösung erreichbare Rauschabstand erhalten bleibt.

Betrachten wir nun den anderen Fall, den der Übersteuerung, d.h. der Überschreitung des vom Wandler vorgegebenen Spannungsbereiches. Im Gegensatz zu analogen Audiosystemen, bei denen eine leichte Übersteuerung nicht unbedingt wahrgenommen wird, da sich die Spannungskennlinie eines analogen Bauteiles oberhalb der Aussteuerungsgrenze nur langsam verändert,

Taken from Amiga-Manuals-Website

haben schon kleine Überschreitungen des Wertebereichs eines A/D-Wandlers fatale Auswirkungen auf das Frequenzbild. Abb. 1.18 zeigt, wie die die Aussteuerungsgrenze überschreitenden Spannungsanteile nicht wie in einem analogen System gedämpft, sondern sauber abgeschnitten werden. Das Ergebnis ist eine Verzerrung in Form von (hauptsächlich ungeradzahligen) Oberwellen, die leider im allgemeinen sehr gut hörbar sind und sich im Klang dementsprechend störend bemerkbar machen.

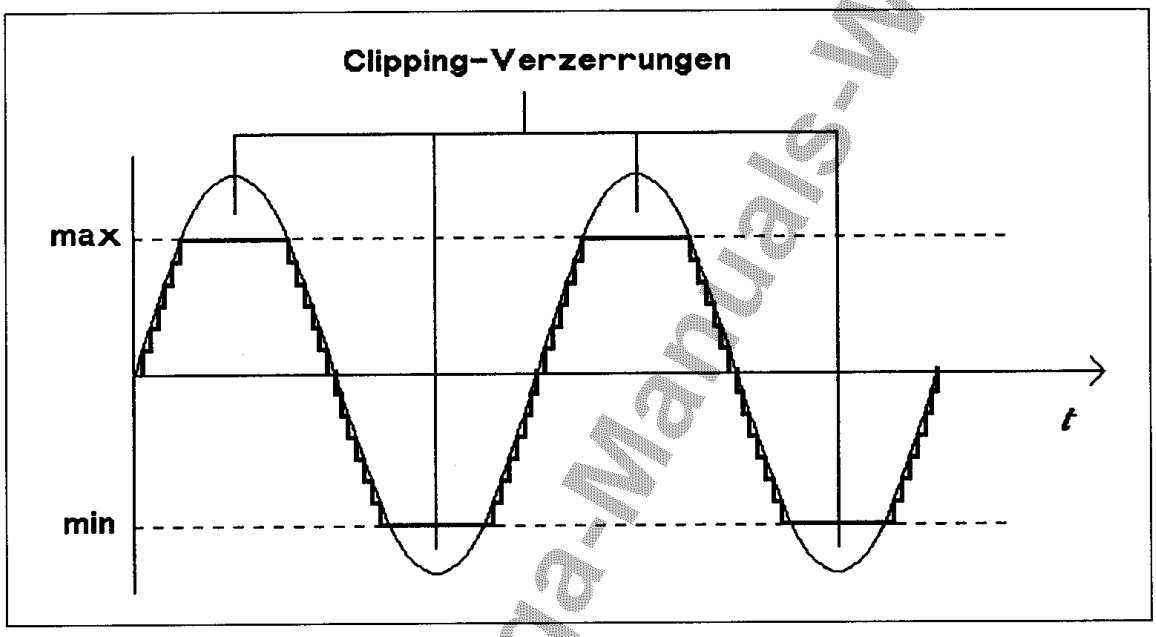


Abb. 1.18: Übersteuerungsverhalten eines A/D-Wandlers

Da in einem akustischen Geschehen sowohl die Intensität als auch die Zeitpunkte des Auftretens von Lautstärkespitzen nicht voraussehbar sind, empfiehlt es sich also, bei der Aussteuerung von A/D-Wandlungsvorgängen grundsätzlich eine gewisse Reserve einzuplanen. Für die Höhe dieser Reserve existieren keine festen Regeln, sie ist im wesentlichen abhängig vom allgemeinen Dynamikverlauf der Musik, also von der Dichte der Stimmenverteilung sowie des Dynamikverhaltens der einzelnen Instrumente. Allerdings sollte man diese Reserve auch nicht zu hoch wählen, da sie in jedem Fall zu Lasten des Signal-/Rauschabstandes geht. Der optimale Wert muß im individuellen Fall stets durch Ausprobieren gefunden werden.

Zeitquantisierung und Shannons-Abtasttheorem

Wie am Beginn des letzten Abschnitts erwähnt, ist für die Wiedergabequalität einer Digitalisierung außer der Wandlerauflösung auch noch die Sampling-Rate von großer Bedeutung, d.h., der zeitliche Abstand zwischen den einzelnen Probenentnahmen. Im folgenden soll untersucht werden, welche Anforderungen an diese gestellt werden müssen, unter welchen Voraussetzungen es also möglich ist, aus einer abgetasteten Wellenform die darin enthaltene Information vollständig zu rekonstruieren.

Taken from Omega-Manuals-Website

Um die Spannungsänderungen zwischen den einzelnen Prüfvorgängen vernachlässigen und die Auswirkungen des Rundungsfehlers besser untersuchen zu können, haben wir im letzten Abschnitt die Sampling-Rate als unendlich hoch angenommen. Da jedoch die Kapazitäten der verwendeten Geräte in der Praxis sehr begrenzt sind (es soll ja auch alles einigermaßen preiswert bleiben), möchte man vor allem in dieser Hinsicht nur die Minimalbedingungen erfüllen müssen. Nicht zuletzt bestimmt hauptsächlich die Sampling-Rate die Menge der pro Zeit anfallenden Datenmengen, was besonders bei Samplern und sonstigen digitalen Musikinstrumenten einen kritischen Punkt bildet. Damit wir jedoch bei der Betrachtung des Einflusses, den die Häufigkeit der Probenentnahme auf die Wiedergabequalität hat, wiederum die Effekte der begrenzten Wertabstufung und Amplitudenquantisierung vernachlässigen können, müssen wir nun die Auflösung unendlich hoch wählen. Dies ist in der Praxis natürlich ebenfalls nicht möglich, hilft aber, die Auswirkungen des Abtastvorgangs auf das Frequenzbild besser von den anderen Störeinflüssen zu unterscheiden.

Abtastung

Wenden wir uns aber nun der Technik der Abtastung selbst zu. Diese ist als der Prozeß einer zyklischen, kontinuierlichen Entnahme von Stichproben zu sehen, wobei der geprüfte Wert im den Zeiträumen zwischen den einzelnen Proben unbekannt ist und ihm deshalb jeweils das letzte Prüfergebnis zugeordnet wird. Eine Abtastung liefert also eine Folge von »Momentaufnahmen« der gemessenen Größe, die ihren Verlauf mehr oder minder genau beschreiben. Diese Genauigkeit ist erstens von der Häufigkeit der Probenentnahmen und zweitens von der Geschwindigkeit abhängig, mit der sich die zu messende Größe ändert.

Um sich diesen Zusammenhang zu verdeutlichen, ist es praktisch, einen zu diesem Phänomen ähnlichen Effekt aus dem Bereich der Optik zu betrachten. Wohl jeder wird schon einmal beobachtet haben, wie im Fernsehen oder Kino sich schnell drehende Räder zeitweise stehenzubleiben oder sich sogar rückwärts zu drehen scheinen. Das von einer Film- oder Fernsehkamera angewandte Verfahren läßt sich insofern mit einer Abtastung vergleichen, als daß ja auch hier in festen zeitlichen Abständen Bilder festgehalten werden, die bei der Reproduktion wegen der Trägheit des Auges wieder zu einem kontinuierlichen Bewegungseindruck verschmelzen. Um diesen Effekt eingehender zu betrachten, machen wir nun einen kleinen physikalischen Versuch. Stellen wir uns hierzu eine Kamera vor, die beispielsweise genau zehn Bilder in der Sekunde macht. Diese richten wir auf eine drehbare Scheibe, die mit einer Markierung versehen ist und deren Umdrehungsgeschwindigkeit wir frei bestimmen können. Um den Effekt direkt zu beobachten, könnte man dasselbe statt mit einer Kamera auch mit einem Stroboskop erzielen, welches die Scheibe im festen Abstand von einer Zehntelsekunde kurz beleuchtet. In Abb. 1.19 sind einige Bildfolgen dargestellt, die die Betrachtung der Scheibe bei verschiedenen Umdrehungsgeschwindigkeiten liefern würde.

Wie im Abschnitt 1.1.2 erläutert, läßt sich eine Kreisbewegung genauso als eine Schwingung mit einer bestimmten Frequenz auffassen. Betrachtet man diese im Vergleich zu der verwendeten »Abtastfrequenz«, hier zehn Lichtblitze in der Sekunde, so läßt sich beobachten, daß die Richtung und wirkliche Geschwindigkeit der Kreisbewegung des Striches auf der Scheibe nur korrekt wahrgenommen wird, solange ihre Frequenz weniger als ungefähr die Hälfte der Frequenz der

Tascam Amiga Manuals Website

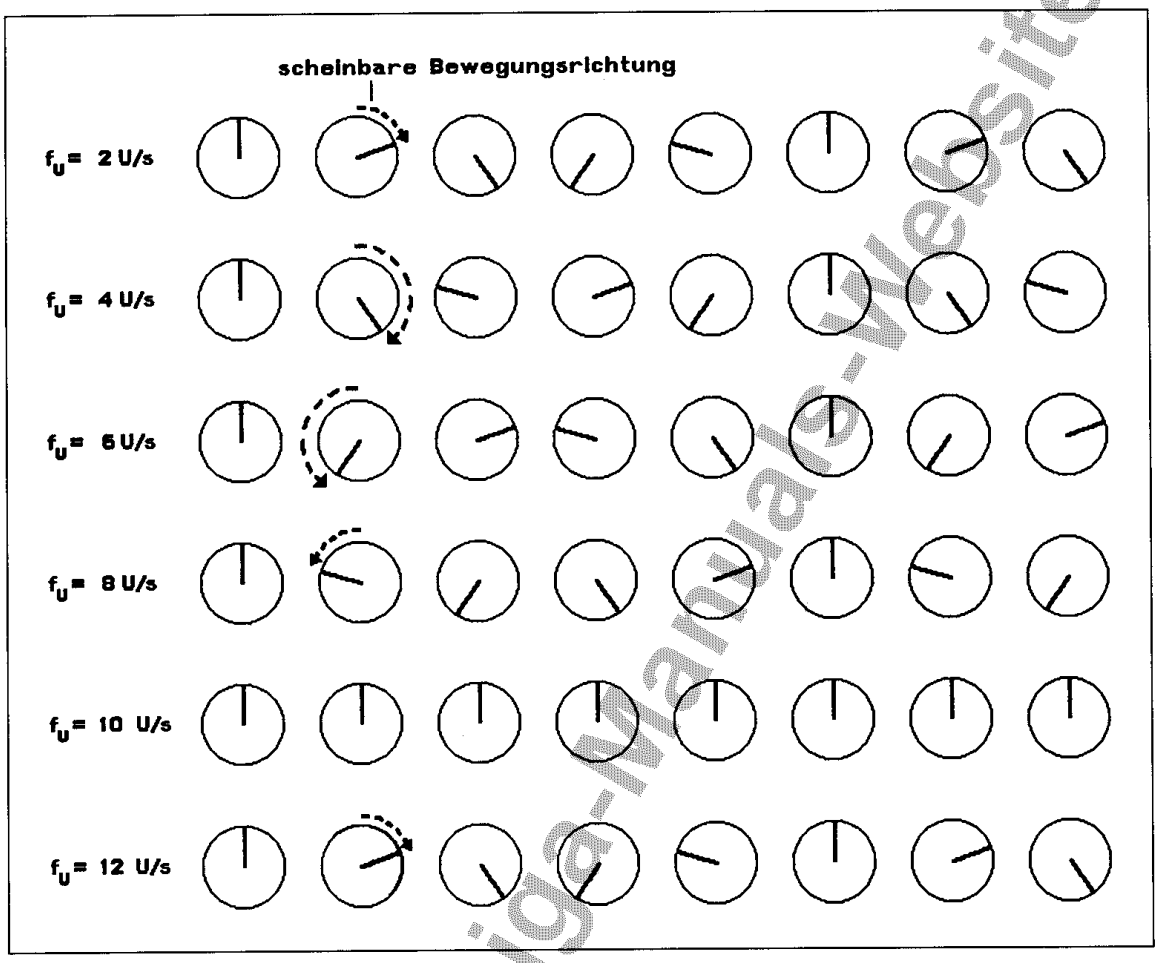


Abb. 1.19: Stroboskopeffekt

Lichtblitze beträgt. Wird sie dagegen größer als eben diese Hälfte, so glaubt das Auge plötzlich eine gegenläufige Rotationsbewegung wahrzunehmen. Steigert man die Umdrehungsgeschwindigkeit der Scheibe weiter, so verlangsamt sich diese scheinbare Gegenbewegung, bis sie bei genauer Gleichheit zwischen Umdrehungs- und Stroboskop-Frequenz ganz zum Stillstand kommt. Bei noch höherer Umdrehungszahl meint man wieder eine Rotation in die richtige Richtung wahrzunehmen, allerdings gegenüber der wirklichen Geschwindigkeit genau um die Stroboskop-Frequenz vermindert, bis sich der oben genannte Effekt abermals bei jedem ihrer Vielfachen wiederholt.

Es scheint also, daß die Abtastfrequenz zur beobachteten Frequenz in einem bestimmten Mindestverhältnis stehen muß, um eine richtige Interpretation der Momentaufnahmen zuzulassen. Ist dies nicht der Fall, so neigt das Auge dazu, nicht die eigentliche, sondern die um die Abtastfrequenz (oder um deren ganzzahlige Vielfache) verminderte Umdrehungsgeschwindigkeit vorzuspiegeln. Hieraus geht auch hervor, daß man bei der Untersuchung solcher Effekte nicht nur von positiven Frequenzen ausgehen darf: Ist in diesem Fall die betrachtete Frequenz immer noch kleiner als die Abtastfrequenz, dann ist ihre Differenz negativ, die scheinbare Drehung hat

Tarn from Amiga-Manuals-Website

also die entgegengesetzte Richtung der Ursprünglichen. Wie wir später noch sehen werden, haben solche »negativen« Frequenzen auch im Audiobereich eine besondere Bedeutung, da sie genauso wie positive Frequenzen wahrgenommen werden und deshalb als in den positiven Bereich »gespiegelt« betrachtet werden müssen.

Mehrdeutigkeiten und Aliasing

Wie wir nun festgestellt haben, kann das Ergebnis der Abtastung einer Schwingung stets auf mehrere Arten interpretiert werden. In unserem Beispiel mit der rotierenden Scheibe haben wir dabei vor allem den Fall kennengelernt, daß anstatt der echten Frequenz diese minus einem ganzzahligen Vielfachen der Abtastfrequenz wahrgenommen wird. Hierbei wählt das Auge wegen seiner Trägheit stets die »bequemste« Interpretation, also dasjenige Vielfache der Abtastfrequenz, welches in der Differenz mit der abgetasteten die kleinste scheinbare Umdrehungsgeschwindigkeit ergibt. Wäre unser Auge jedoch »schnell« genug, um noch höhere Bewegungsgeschwindigkeiten genau zu sehen, so könnten wir sehr viele Interpretationen gleichzeitig wahrnehmen und zwischen ihnen »hin- und herschalten«, und zwar sähen wir dabei sowohl die Differenz als auch die Summe der Drehfrequenz mit den ihr benachbarten Vielfachen der Abtastfrequenz.

Jetzt wollen wir jedoch unsere Gedanken in den Audiobereich übertragen. Betrachten wir also eine von einem A/D-Wandler abgetastete Wellenform, wobei wir der Einfachheit halber eine reine Sinusschwingung wählen, um uns nicht auch noch mit mehreren Frequenzkomponenten herumärgern zu müssen. Wenn wir feststellen wollen, welche Interpretationen die durch die Abtastung erhaltene Folge von Werten zuläßt, so müssen wir nach den Ergebnissen des obigen Versuches die Differenz und die Summe der untersuchten Frequenz und aller ganzzahligen Vielfachen der Abtastfrequenz bilden, also:

$$f_n = n * f_A \pm f_0$$

mit $n = 1,2,3,...$
f_n - Ergebisfrequenzen
f_0 - Eingangsfrequenz
f_A - Abtastfrequenz

Formel 16

In Abb. 1.20 ist die Abtastung einer Wellenform mit den sich außerdem anbietenden Interpretationen dargestellt. Es ist zu erkennen, daß überall dort Mehrdeutigkeiten in einer Folge von Momentaufnahmen auftreten, wo die jeweiligen Amplituden von verschiedenen Frequenzen jeweils zum Zeitpunkt der Abtastung ein und denselben Wert aufweisen. Tatsächlich ergibt die Fourier-Analyse einer abgetasteten Schwingungsfunktion exakt diese Spektralverteilung, also genau die Differenzen und Summen der Eingangsfrequenz und den Vielfachen der Abtast-

frequenz. Diese »verfremdenden« (engl. aliasing) Schwingungen werden vom menschlichen Gehör als besonders störend wahrgenommen, da sie in der direkten Umgebung der Eingangsfrequenz noch ziemlich laut sind und überdies oft nicht einmal in einem ganzzahligen Verhältnis zur dieser stehen, weshalb sie meist sehr unharmonisch und »schräg« klingen.

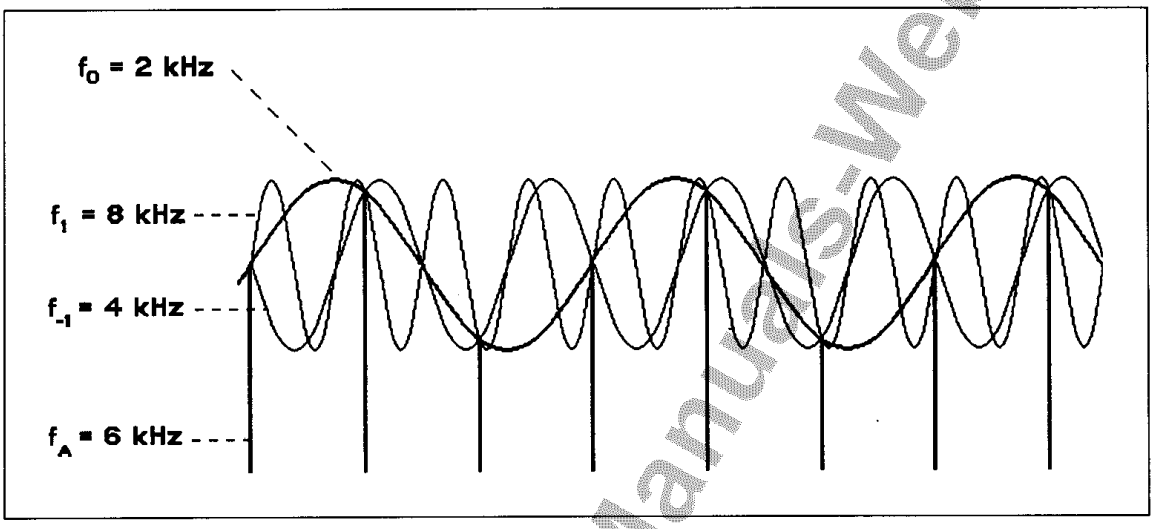


Abb. 1.20: Mehrdeutigkeiten der Abtastfolge einer Schwingung

Eine Eingangsfrequenz f(0) erhält also durch eine Abtastung mit der Häufigkeit f(A) einen ganzen »Kranz« von zusätzlichen Frequenzen, jeweils im Abstand von f(A), sowohl in positiver als auch in negativer Richtung. Wie jedoch oben erwähnt, müssen alle negativen Frequenzen genau wie positive betrachtet werden, da das Gehör zwischen ihnen keinen Unterschied macht. Man muß sich also alle Aliasingfrequenzen, die durch eine Abtastung auftreten, an der 0-Achse in den positiven Bereich gespiegelt vorstellen, um das für den Audiobereich relevante Frequenzbild zu erhalten (siehe Abb. 1.21).

Abtastrate und Übertragungsbereich

Im Audiobereich möchte man in aller Regel einen begrenzten Frequenzbereich, von 0 bis zu einer bestimmten Maximalfrequenz (auch Übertragungs- oder Nutzbereich genannt), korrekt und fehlerfrei übertragen. Im Fall der Digitalisierung durch Abtastung lautet also die Bedingung hierfür, daß in genau diesem Bereich, sowohl auf der positiven wie auf der negativen Seite, keine Aliasing-Frequenzen auftreten dürfen, da diese sonst unser gewünschtes Signal mit einem Fehler verunreinigen, der nicht mehr korrigierbar ist. Um dies zu erreichen, muß man sich die Frage stellen, welche Höchstgrenze für die Eingangsfrequenzen bei einer festgelegten Samplingrate, bzw. welche Samplingrate bei einem festgelegten Übertragungsbereich veranschlagt werden muß, damit keine Aliasing-Frequenzen unterhalb der maximal zu übertragenden Frequenz zu liegen kommen. Bei der Spiegelung ist zu beobachten, daß genau dann keine Spiegelungsfrequenzen im Nutzbereich vorhanden sind (was ja die Bedingung für eine originalgetreue Wiedergabe bedeutet), wenn die Abtastfrequenz mindestens so groß ist wie die Summe aus der

Taken from Amiga-Manuals-Website

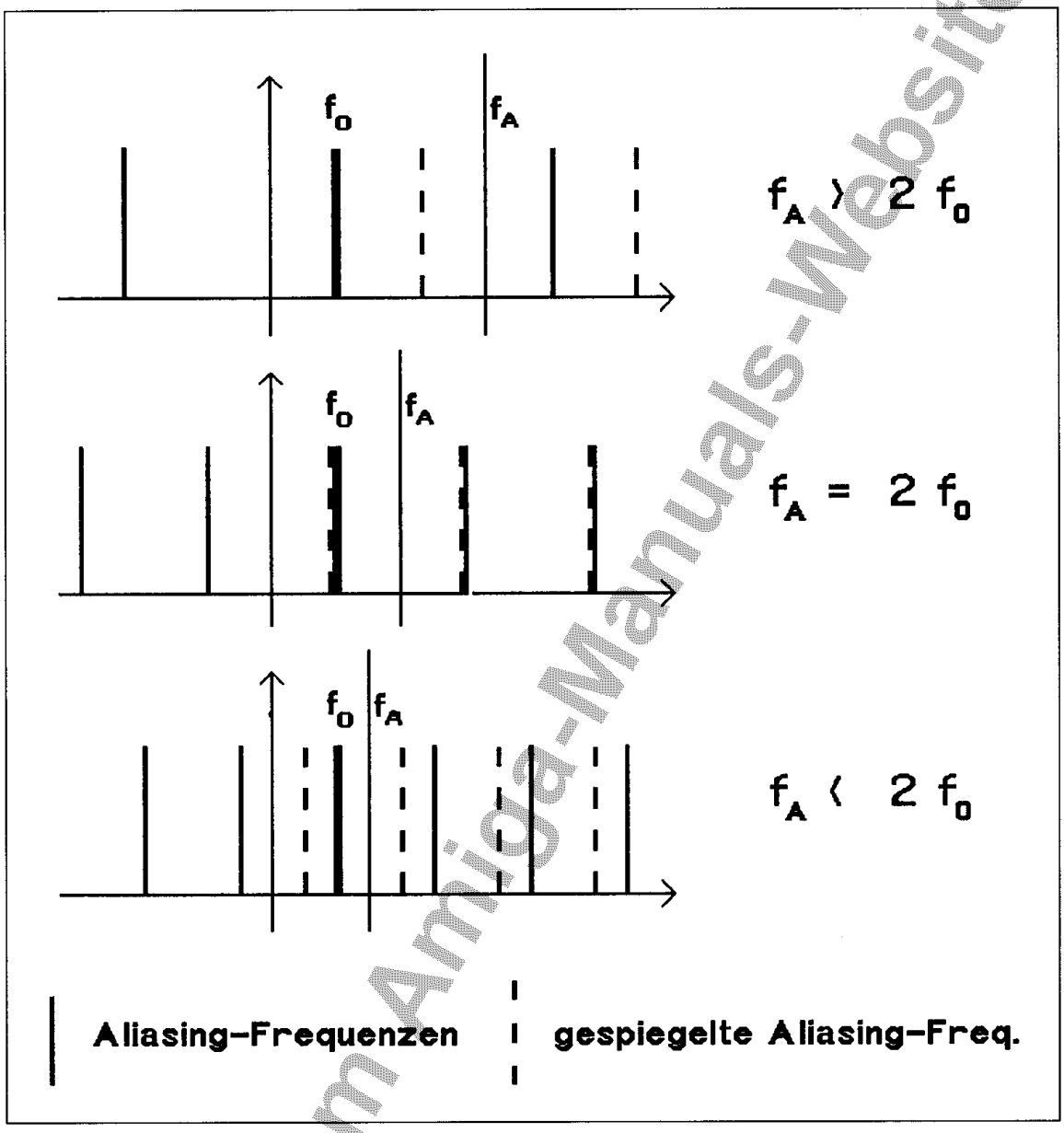


Abb. 1.21: Aliasing- und Spiegelungsfrequenzen bei verschiedenen Abtastraten

maximalen Eingangsfrequenz und der höchsten Frequenz des Nutzbereiches. Für den Fall, daß diese beiden Größen gleich sind, läßt sich auch sagen, daß die Samplingrate mindestens das doppelte der maximalen Eingangsfrequenz betragen muß.

Diese Beziehung leitet sich folgendermaßen her: Da bei einer Frequenz, ob positiv oder negativ, nur der absolute Betrag zählt, muß die Grenzfrequenz des Nutzbereiches, d.h. die höchste zu übertragende Frequenz, der folgenden Ungleichung genügen:

Taken from Amiga-Manuals-Website

$$f_{nutz} < | n*f_A \pm f_{ein} |$$

wobei
- f_{nutz} - maximale Nutzfrequenz
- f_{ein} - maximale Eingangsfrequenz
- f_A - Samplingrate
- n = 1,2,3,...
- $| \; |$ - absoluter Betrag

Formel 17

Weil wir jedoch im allgemeinen davon ausgehen können, daß die maximale Eingangsfrequenz zumindest kleiner ist als die Samplingrate selbst (andernfalls erhält man sowieso einen hoffnungslosen Wellensalat), und somit die Ergebnisse der rechten Seite bei einem höheren n immer größer sind als bei kleinen n, kann man es in der Ungleichung fortlassen:

$$f_{nutz} < | n*f_A \pm f_{ein} |$$

Formel 18

Aus dem gleichen Grund ist auch der Betrag der Summe stets höher als der der Differenz, weshalb man auch das Pluszeichen streichen kann

$$f_{nutz} < | f_A - f_{ein} |$$

Formel 19

und darüber hinaus, da das Ergebnis der Differenz jetzt immer negativ ist

$$| f_A - f_{ein} | = f_A - f_{ein}$$

Formel 20

lassen sich nun auch die Betragstriche weglassen

$$f_{nutz} < f_A - f_{ein}$$

Formel 21

Dies führt zu der folgenden Anforderung an die Abtastfrequenz

$$f_A > f_{nutz} + f_{ein}$$

Formel 22

oder, wenn wie erwähnt die maximale Eingangsfrequenz mit der Nutzbandbreite identisch sein soll:

$$f_A > 2 * f_{nutz}$$

Formel 23

Dies ist auch die Aussage des Shannonschen Abtasttheorems: *Ein beliebiges, in seiner spektralen Verteilung begrenztes Signal mit der Grenzfrequenz f(nutz) läßt sich aus einer durch Abtastung mit der Frequenz f(A) erzeugten Folge von Werten genau dann wieder vollständig rekonstruieren, wenn die Bedingung f(A)>=2*f(nutz) erfüllt ist.* Ist dies nicht der Fall, so werden diejenigen Frequenzen im Spektrum des Eingangssignals, die über der Hälfte Abtastrate liegen, in den Nutzbereich gespiegelt, wodurch dieser zum Teil auf nicht mehr korrigierbare Weise verunreinigt wird. Dieses Verhalten ist in Abb. 1.22 noch einmal genau dargestellt.

Anti-Aliasing-Filter

Dies hat in der Praxis der Audiotechnik vor allem eine Konsequenz: Angenommen man möchte ein bestimmtes Signal, beispielsweise den Ton einer Gitarre, durch einen A/D-Wandler digitalisieren und anschließend mit einem Sampler wiedergeben. Ist man hierbei aber in der Wahl der Abtastrate so beschränkt, daß die oben genannte Bedingung nicht erfüllt werden kann, wird auf jeden Fall ein Teil des hörbaren Spektrums verzerrt und eine korrekte Aufnahme unmöglich. Abhilfe schafft hier nur ein Frequenzfilter, der für alle Schwingungen oberhalb der erforderlichen

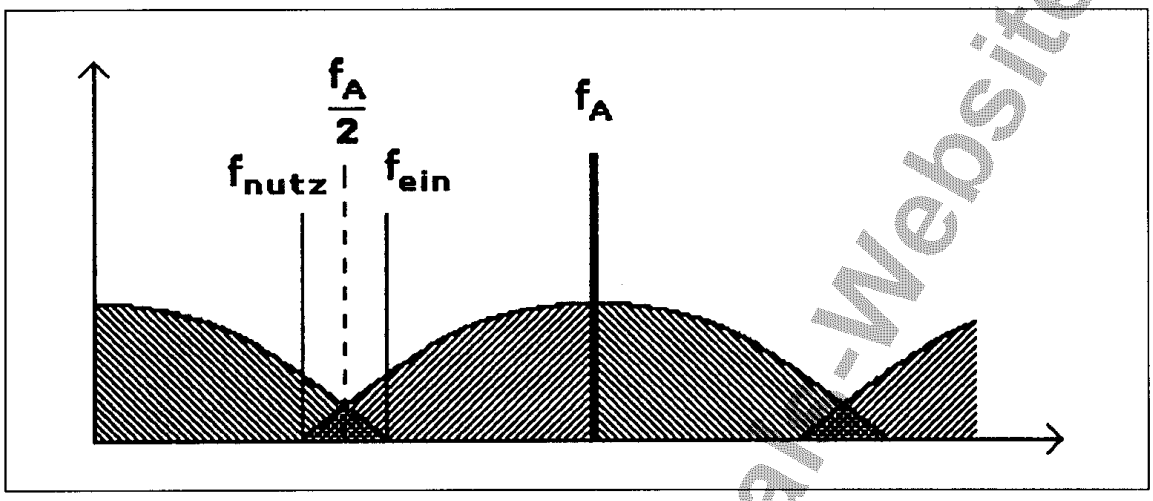


Abb. 1.22: Rekonstruktionsfehler

Grenzfrequenz undurchlässig ist und so das Spektrum auf die gerade noch verkraftbaren Anteile »zurechtstutzt«. Einen solchen Filter bezeichnet man auch als Tiefpaß-Filter, da er nur Tonhöhen unter dieser gewissen Grenze »passieren« läßt (Abb. 1.23). Nur mit Hilfe einer solchen Tiefpaßfilterung kann bei der Wiedergabe das ursprüngliche Signal in reiner Form rekonstruiert werden.

Ein solches Filter muß darüber hinaus noch einigen Ansprüchen genügen:

❑ eine hohe Dämpfung im Sperrbereich – damit die Frequenzanteile oberhalb der gewünschten Grenze mit Sicherheit unhörbar sind und so auch nicht in den Nutzbereich gespiegelt werden können, müssen ab dieser Grenze alle Frequenzen soweit abgeschwächt werden, daß ihre Amplitude möglichst unter der Auflösung des verwendeten Wandlers liegt;

❑ einen steilen Übergang vom Durchlaß- zum Sperrbereich – ist dieser Übergangsbereich zu breit, so geht entweder zuviel von der Information im Nutzbereich verloren, oder die Sperrdämpfung ist an der kritischen Frequenz noch nicht groß genug, so daß doch noch Spiegelungsfrequenzen auftreten. Deshalb muß das Filter eine sehr steile Charakteristik aufweisen.

❑ Die Eigengeräusche (thermisches Rauschen) und der Klirrfaktor des Filters sollten auf jeden Fall kleiner sein als das durch die Auflösung bedingte Quantisierungsrauschen.

Verfügt man über keine speziellen Vorrichtungen, wie z.B. einen speziellen variablen Filter, der für diese Zwecke ausgelegt ist, wird man sich im Hobbybereich wohl mit einem möglichst vielbändigen Equalizer oder im Notfall auch mit der normalen Klangregelung einer Stereoanlage einigermaßen behelfen können.

Folgerungen

Wenn wir nun unsere bisher gewonnenen Erkenntnisse zusammenfassen, so läßt sich für die Praxis des Digitalisierens eines Audiosignals grundsätzlich das folgende Verfahren empfehlen:

Taken from Amiga-Manual-Website

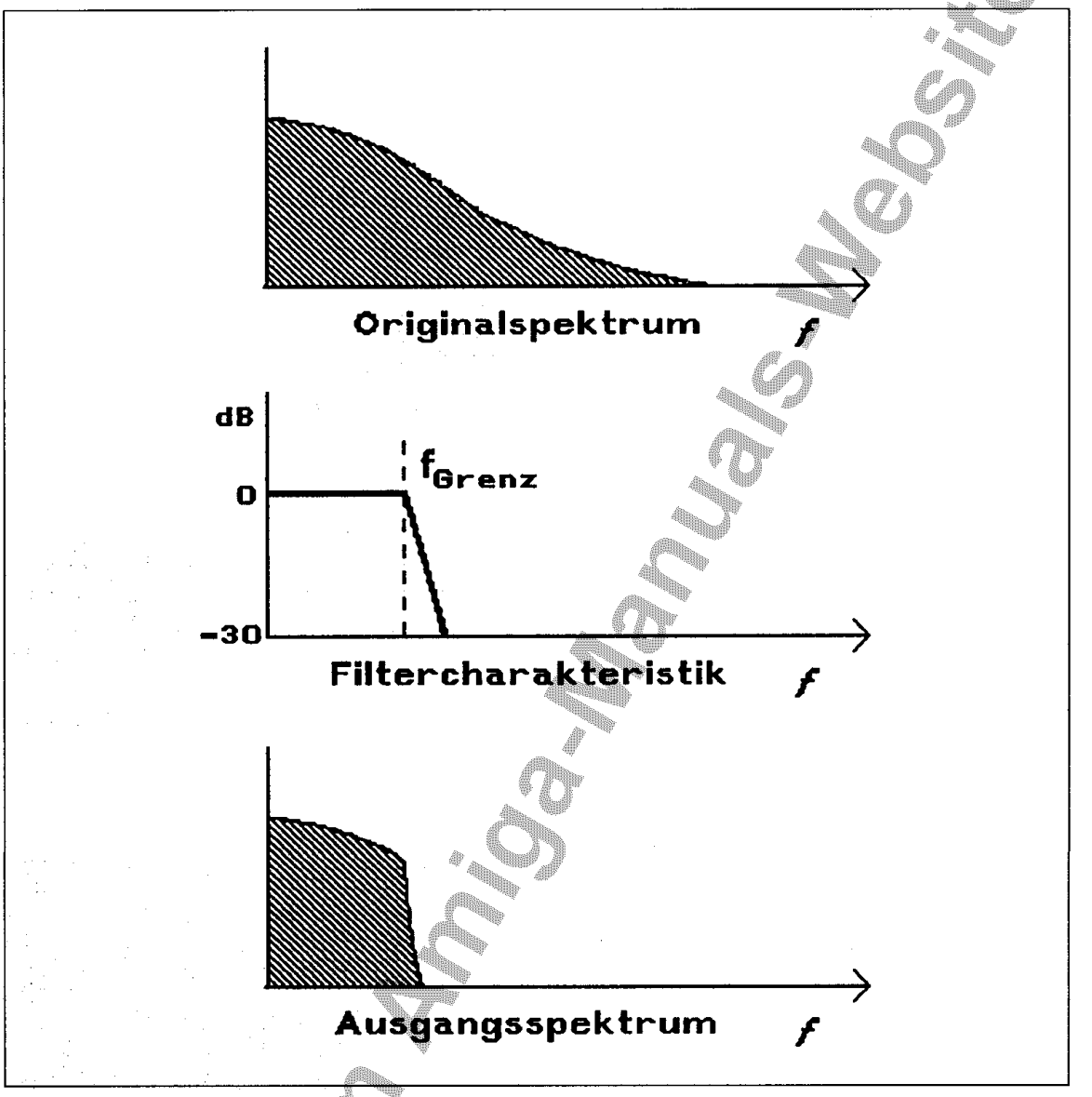


Abb. 1.23: Tiefpaßfilter

Zuerst einmal müssen wir den Frequenzbereich des Eingangssignals bestimmen, das wir digitalisieren wollen. Da wohl nicht jeder einen Spektralanalysatoren sein eigen nennt, ist dies bei Klängen mit komplexeren Spektralverläufen (wie aus einer Stereoanlage) oft sehr schwierig genau zu bestimmen, weshalb man hier von vornherein eine Frequenzbegrenzung durch einen Filter vornehmen sollte. Bei einer leichter zu bestimmenden Klangquelle jedoch, wie einem Musikinstrument, läßt sich der Frequenzbereich auch meist aus der Grundfrequenz, unter Berücksichtigung der instrumentspezifischen Oberwellen, mit guter Näherung bestimmen.

Als nächstes müssen wir uns die Frage stellen, ob wir den im ursprünglichen Signal enthaltenen Frequenzbereich überhaupt voll nutzen wollen. Möchten wir bei der Wiedergabe auf keine der

Taken from Amiga-Manuals-Website

Komponenten unseres Signals verzichten, so müssen wir die maximale Nutzfrequenz gleich der höchsten Eingangsfrequenz annehmen. In den meisten Fällen ist man hierbei aber an die maximale Ausgangsfrequenz des Wiedergabegerätes gebunden. Hat man, wie z.B. im Amiga, bei der Ausgabe nur Tonhöhen bis höchstens 4 kHz zur Verfügung, so hat es keinen Sinn, den Nutzbereich bei der Aufnahme wesentlich höher zu wählen. Im allgemeinen kann man also ruhig die maximale Frequenz, die bei der Wiedergabe möglich ist, als die Grenze des Übertragungsbereiches ansehen.

Sind diese beiden Größen festgestellt, so kann man darangehen, die für diese Rahmenwerte benötigte Abtastrate zu ermitteln. Dazu muß man, wie gezeigt, lediglich die maximale Eingangs- und Nutzfrequenz zueinander addieren, bzw. bei Gleichheit eine von ihnen verdoppeln. Die so erhaltene Frequenz ist die Abtastrate, die mindestens eingehalten werden muß, um die Bedingungen des Shannon-Theorems zu erfüllen.

Sollte es aber entweder nicht möglich oder nicht praktikabel sein, das Signal mit dieser Frequenz abzutasten, z.B. weil die hierbei anfallende Datenmenge zu groß wäre, so ist man in jedem Fall zu einer weiteren Begrenzung des Eingangsspektrums durch einen Filter gezwungen. Dann ist es empfehlenswert, zuerst die größtmögliche Abtastrate zu bestimmen. Diese teilt man anschließend durch zwei, was dann die Obergrenze sowohl für den Eingangs- als auch für den Nutzbereich darstellt. Wenn man nun die Sperrfrequenz des Filters auf diesen Bereich einstellt, kann bei der Wiedergabe stets ein sauberes, störungsfreies Signal rekonstruiert werden.

1.2.2 Wiedergabe

Möchte man nun die durch digitale Wertefolgen repräsentierten Audiosignale wieder hörbar machen, so müssen sie in irgendeiner Form rekonstruiert, d.h. wieder in analoge Spannungen rückgeführt werden. Wie in Abb. 1.11 am Anfang dieses Abschnitts dargestellt, ist der sog. D/A-Wandler das hierfür zuständige Glied der Komponentenkette. Dieser hat die Aufgabe, einen am Digitaleingang anliegende Kombination von Bit-Werten auf Befehl in eine dazu proportionale Spannung umzuwandeln. Wie bei der A/D-Wandlung, so wird auch für diesen Vorgang eine bestimmte Zeit benötigt. Wird nämlich der Befehl zur Auswertung des gerade anliegenden Wertes erteilt, so kann es vorkommen, daß einige Bits ihren neuen Zustand schneller einnehmen als andere, da sich z.B. die Übergangszeit von einem 0-Wert auf 1 fast immer von der Dauer des umgekehrten Vorgangs unterscheidet. Dies führt zu einer Phase, in der die Spannung undefinierte Werte aufweist, sog. transiente Störspannungen. Um dieses Problem zu umgehen, bedient man sich auch hier eines Sample&Hold-Verstärkers. Sobald feststeht, daß der Wandler seine »Erholungsphase« hinter sich hat, greift dieser die nun konstante Spannung ab und legt sie auf seinen Ausgang, bis die Anordnung in den nächsten Wandlungszyklus eintritt.

Am Ausgang des S&H-Verstärkers liegt nun also eine ziemlich saubere Treppenspannung an. Diese enthält jedoch, wie wir in den vorangegangenen Abschnitten gesehen haben, noch wesentlich mehr Frequenzanteile als das von uns aufgenommene Originalsignal. Wir wissen nun aber, welche Eigenschaften die bearbeiteten Audiodaten haben müssen, damit die gewünschten Teile des Signals von den durch die Abtastung entstandenen zusätzlichen Frequenzen korrekt getrennt werden können. Hierfür schließt sich an den S&H-Verstärker noch ein Filter an, der die Treppenfunktion glättet und so die Aliasing-Frequenzen entfernt, um endlich dem Signal seine ursprüngliche Form wiederzugeben.

Tomas Amiga-Manuals Website

Beschäftigen wir uns jedoch zunächst mit dem D/A-Wandler selbst. Dieser besteht im wesentlichen aus einer Reihe von Widerständen, deren Werte mit den Stellenwerten der einzelnen Bits korrespondieren. Der Strom bzw. die Spannung an den Widerständen wird durch das jeweils zugehörige Datenbit ein- oder ausgeschaltet und die Summe dieser Spannungen auf einen Sammelpunkt zusammengeleitet, wodurch das Ergebnis dem zur Zeit anliegenden digitalen Wert entspricht. Einen nach diesem Verfahren arbeitenden Wandler nennt man auch einen parallelen D/A-Wandler.

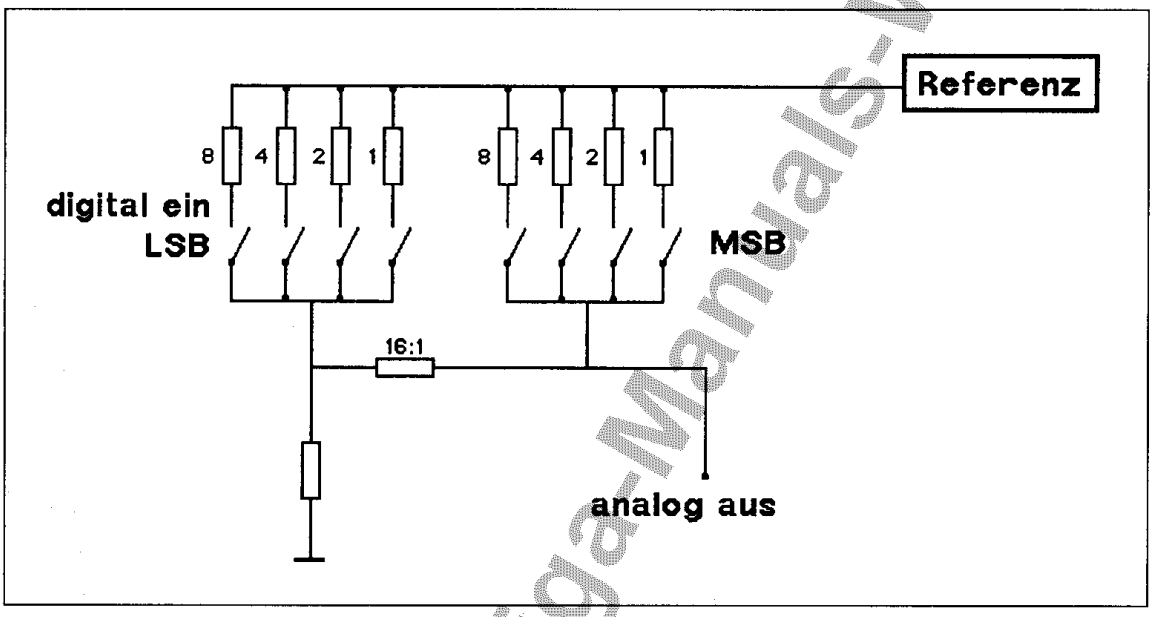


Abb. 1.24: Paralleler D/A-Wandler

Abb. 1.24 zeigt einen solchen Wandlertyp. Wie bei den A/D-Wandlern auch, so gibt es auch hier Typen mit rein postiver Darstellung von 0 bis 2^n-1, und solche die das Zweierkomplement verwenden. Diese Darstellungsart ist bei der obige Anordnung ganz einfach dadurch zu verwirklichen, daß dem höchstwertigen Bit (MSB) anstatt einem positiven ein negativer Spannungswert zugeordnet wird, der in seinem Absolutwert dem normalen Wert des MSB, also der halben Darstellungsbreite entspricht. Dadurch wird die Ausgangsspannung analog zum Zweierkomplement in den negativen Bereich »heruntergezogen«.

In Abb. 1.24 ist ebenfalls zu erkennen, daß die transienten Störspannungen auf jeden Fall größer als 1 LSB sind, da sie den Werten der Bits entsprechen, die zu spät auf ihrem neuen Wert »ankommen«, und deshalb nahezu jede beliebige Stärke bis 1 MSB annehmen können. Dies macht die Verwendung eines S&H-Verstärkers zwingend erforderlich. Meist ist dieser jedoch schon in einem D/A-Wandler integriert, so daß sich der Anwender keine Gedanken mehr um die »Sauberkeit« des Ausgangssignals machen muß.

Kommen wir nun zum beinahe wichtigsten Teil des Vorgangs, der Filterung, d.h. der Aussiebung des Originalsignals aus dem Frequenzgemisch, welches die von einem D/A-Wandler gelieferte

Take from Amiga-Manuals-Website

Treppenspannung darstellt. Wie aus den Abschnitten über die Abtastung bekannt ist, mußten ja schon bei der Aufnahme für die vollständige Rekonstruierbarkeit einige Voraussetzungen erfüllt sein: Die Abtastrate f(A) mußte mindestens das Doppelte der höchsten Eingangsfrequenz f(ein) betragen oder, falls nicht der ganze Eingangsbereich genutzt werden sollte, mindestens so groß sein wie die Summe aus dieser und der höchsten hörbaren Frequenz f(nutz). Bei der Aufnahme wurde durch die Wahl der Größen f(A) und f(ein) der nutzbare Bereich endgültig festgelegt, da sich oberhalb von ihm ja bereits die ersten Störanteile befinden. Die Grenze dieses Bereiches, die sich demzufolge aus der Differenz f(A) – f(ein) ergibt, bildet auch die Grenzfrequenz, die der Filter für die korrekte Abtrennung der mit Aliasing-Frequenzen verunreinigten Bereiche aufweisen muß. Die sonstigen Ansprüche, die an einen solchen Filter gestellt werden müssen, sind ähnlich denen der bei der A/D-Wandlung eingesetzten Anti-Aliasing-Filter:

- ❑ große Filtersteilheit
- ❑ hohe Sperrdämpfung
- ❑ Verzerrungen und Eigengeräusche geringer als 1 LSB

Da uns in diesem Zusammenhang jedoch hauptsächlich der Bereich der Instrumentalanwendungen, sprich der Samplertechnik interessiert, wollen wir im folgenden von den hier herrschenden Voraussetzungen ausgehen. Die Tiefpaßfilter, die in Sampler-Geräten wie auch dem Amiga installiert sind, sind im allgemeinen keine variablen Filter, sondern haben eine fest eingestellte Sperrfrequenz. Diese stellt letztendlich die Grenze des Nutzbereichs dar. Das hat auf die oben beschriebene Wahl der anderen Parameter die folgenden Konsequenzen: Erstens darf in der obigen Berechnung der Nutzbereich nicht kleiner gewählt werden als die Tiefpaßgrenze, da sonst die Aliasing-Frequenzen hörbar würden, die sich unterhalb von ihr befinden. Zweitens hat es aber auch keinen Sinn, den Nutzbereich höher anzusetzen als die durch den Filter gegebene Grenze, weil der Bereich darüber niemals hörbar und deshalb die dafür nötige Abtastrate unnötig hoch wäre. Hieraus ist zu erkennen, daß die Sperrfrequenz dieses Filters eine wichtige Rahmenbedingung für sämtliche hier besprochenen Vorgänge darstellt und die anderen Faktoren jeweils nach Möglichkeit darauf eingestellt werden sollten.

Im Gegensatz zu den reinen Aufnahme- und Wiedergabetechniken, bei denen die digitalisierten Audiosignale stets mit der gleichen Geschwindigkeit abgespielt werden, mit der sie aufgenommen wurden, muß man im Bereich digitaler Musikinstrumente von variablen Wiedergaberaten ausgehen. Tatsächlich ist es gerade das Prinzip der Sampling-Technik, bei einer bestimmten Abtastrate aufgenommene Signale mit verschiedenen Datenausgaberaten wiederzugeben, um deren Frequenz und Tonhöhe frei variieren zu können. Das beschert uns natürlich einen ganz neuen Aspekt für die Wahl der Parameter f(A), f(ein) und f(nutz): Soll die digitalisierte Wellenform auch mit einer geringeren Datenrate abgespielt werden, als bei der Aufnahme verwendet wurde, so verschieben sich die Grenzen für Abtastrate und Eingangsbereich entsprechend.

In Abb. 1.25 sind drei verschiedene Situationen bei der Wiedergabe eines digitalisierten Signals zu erkennen: Im Fall a) ist die Wiedergaberate bzw. die Aufnahmerate und der sich daraus ergebende Abstand zwischen Nutz- und Aliasing-Frequenzen zu niedrig gewählt. Dies hat zur

Tarentola Amiga Manuals - Mirror

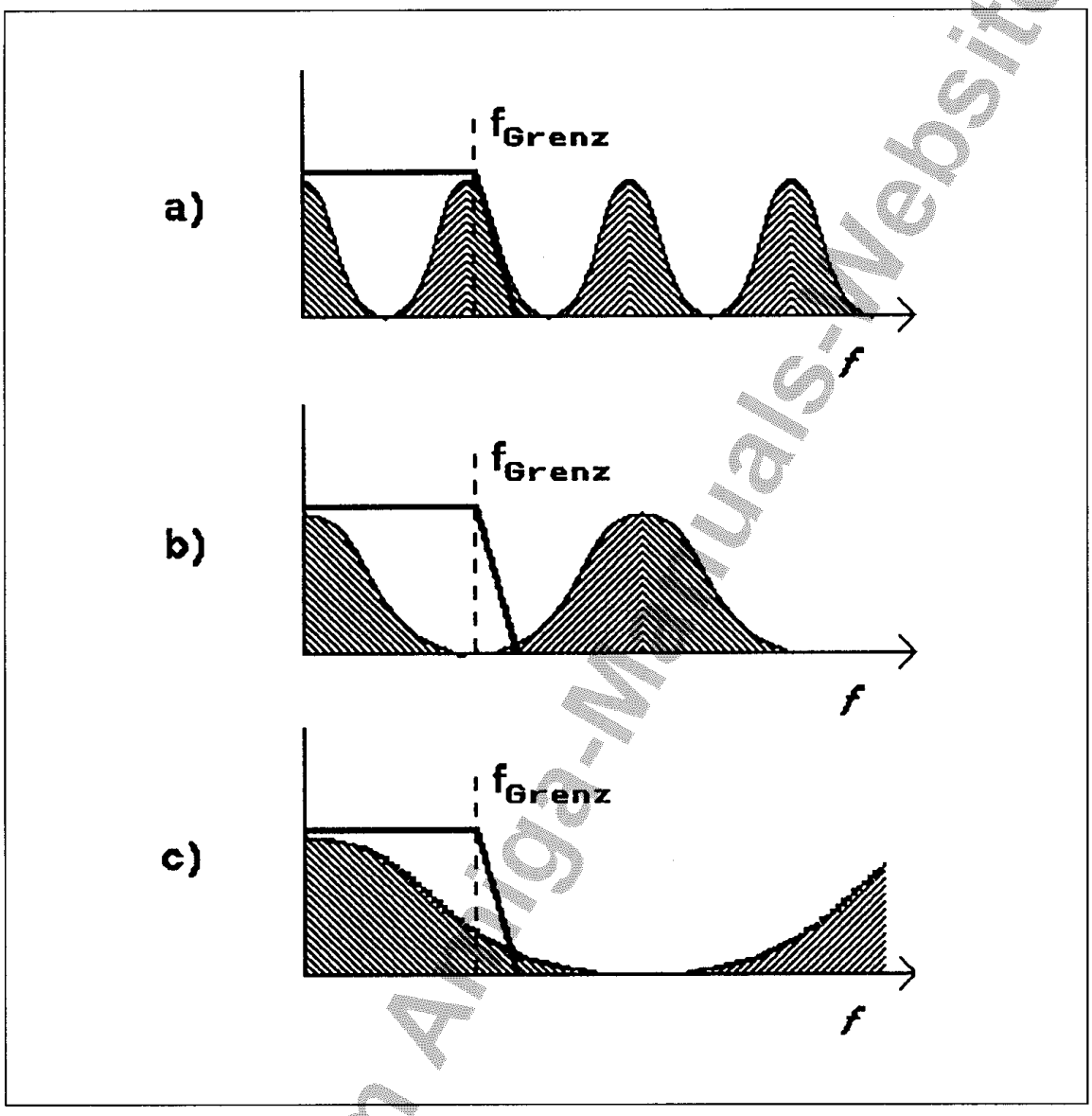


Abb. 1.25: Wiedergabe eines Audiosignals bei verschiedenen Datenraten

Folge, daß diese Störkomponenten unter die Filterfrequenz fallen und sich deshalb im Klangbild unangenehm bemerkbar machen. Der Fall b) stellt den Idealfall dar. Hier sind die Störanteile sämtlich durch die Filterwirkung unhörbar, der Nutzbereich hingegen befindet sich aber noch ganz unterhalb der Sperrfrequenz und ist deshalb noch voll wahrnehmbar. Im Fall c) treten zwar auch keine Aliasing-Frequenzen mehr auf, dafür wird aber ein Teil des Nutzbereiches vom Filter verschluckt, was zu einem gewissen Verlust an Information im hörbaren Bereich führt.

Wenn wir also auch mit Abspielraten unterhalb der Aufnahmerate arbeiten wollen, müssen wir die Bedingungen für eine störungsfreie Wiedergabe folgendermaßen umformulieren: Die Grenze des

Taken from Amiga-Manuals-Website

Nutzbereichs, f(nutz) = f(A) – f(ein), geteilt durch das Verhältnis zwischen Aufnahme- und Wiedergaberate, darf nicht kleiner werden als die Grenzfrequenz des Filters. Oder formell ausgedrückt:

$$(f_A - f_{ein}) * \frac{f_W}{f_A} > f_{Grenz}$$

mit

f_A - Samplingrate
f_W - Wiedergaberate
f_{ein} - maximale Eingangsfrequenz
f_{Grenz} - Grenzfrequenz des Filters

Formel 24

Die Audiohardware des Amiga

2

Der Amiga wird nicht zuletzt wegen seiner Soundfähigkeiten als Computer einer neuen Generation bezeichnet. Während bis zu seinem Erscheinen nur Home- und Personalcomputer mit relativ simplen Soundchips im Handel waren, meist mit solchen aus älteren Videospielautomaten wie z.B. beim Atari ST, kann sich der Anwender des Amiga sämtlicher Möglichkeiten und Vorteile der Sampling-Technik bedienen.

Wie auch seine Grafikfähigkeiten, so erhält der Amiga seine Fähigkeit zur Musikwiedergabe durch die speziell für ihn entwickelten Spezialchips Denise, Agnus und Paula. Im neuesten Modell, dem Amiga 3000 sind die beiden ersteren zu Super-Denise und Super-Agnus weiterentwickelt worden sowie neue Spezialchips (Amber, Ramses) hinzugekommen. In diesem Kapitel werden wir uns jedoch nur mit Agnus und Paula beschäftigen.

Derjenige der Customchips, der letztendlich das analoge Tonsignal liefert, ist Paula; in diesem befinden sich die digitalen Klangerzeugungs-Einheiten, die D/A-Wandler. Es stehen insgesamt vier Wandler zur Verfügung, wobei jeweils zwei zu einem der beiden Stereokanäle des Amiga zusammengefaßt werden. Wie wir später noch genauer sehen werden, gibt es zwei Methoden, um die D/A-Wandler mit Daten zu versorgen:

❑ Prozessor-Betrieb – In dieser Betriebsart versogt der Prozessor die D/A-Wandler selbst mit den benötigten Audiodaten. Hierbei werden in einem von der Audioeinheit ausgelösten Interrupt jeweils zwei Sampling-Werte zugleich übergeben, die nacheinander ausgegeben werden. Da dies dem Prozessor sehr viel Rechenzeit wegnimmt, sollte dieser Modus nur in Fällen angewandt werden, in denen die Verarbeitung von Audiodaten in Echtzeit verlangt wird (z.B. direkte Wiedergabe von einem angeschlossenen Digitalisierer).

❑ DMA-Betrieb – Beim DMA-Betrieb (DMA = Direct Memory Access, direkter Speicherzugriff) werden die D/A-Wandler von der DMA versorgt, die auch für die Datenübertragung von Disk, die Bilderzeugung und den Blitter zuständig ist. Die gesamte DMA-Steuerung wird vom Agnus-Chip erledigt, der außerdem noch den Videocoprozessor (Copper) sowie den erwähnten Blitter enthält. Hierbei wird die »Aufmerksamkeit« des Prozessors nur dann verlangt, wenn die gerade abgespielte Datensequenz zu Ende ist und neue Einstellungen benötigt werden. Dies hat gegenüber der obigen Methode natürlich den Vorteil immenser Rechenzeitersparnis. Es ergeben sich aber auch hier einige systembedingte Einschränkungen, vor allem hinsichtlich der Wiedergaberate.

In diesem Kapitel sollen zunächst der Aufbau und die Eigenschaften der beiden für die Klangerzeugung verantwortlichen Customchips erläutert werden, wobei speziell die durch die Hardwarearchitektur gesetzten Grenzen der Wiedergabequalität sowie die Eigenschaften der Rekon-

struktionsfilter genauer beleuchtet werden. Anschließend folgt ein Abschnitt über die verschiedenen Anwendungsarten der Audiohardware, in dem insbesondere der Aufbau und die Programmierung der Hardwareregister behandelt werden sollen.

2.1 Die Customchips

2.1.1 Paula – die D/A-Wandlerin

Wie in Abb. 2.1 zu sehen, verfügt Paula außer der Kontrollogik für Disk u.ä. über vier D/A-Wandler, von denen jeweils zwei zusammen auf einen der Audioausgänge des Amiga geschaltet sind. Hierbei bilden die Wandler mit der Nummer 0 und 3 den linken, Wandler 1 und 2 den rechten Stereokanal. Damit jeder dieser D/A-Wandler in Lautstärke und Frequenz frei modulierbar ist und außerdem die wiederzugebenden Daten frei gewählt werden können, besitzt jeder Wandler einen Satz von Hardwareregistern, in die der Prozessor die jeweiligen Parameter übergeben kann. Diese werden von der Audio-Kontrollogik aus den Registern übernommen und ausgewertet, die dann die entsprechenden DMA- oder Interruptanforderungen ausführt.

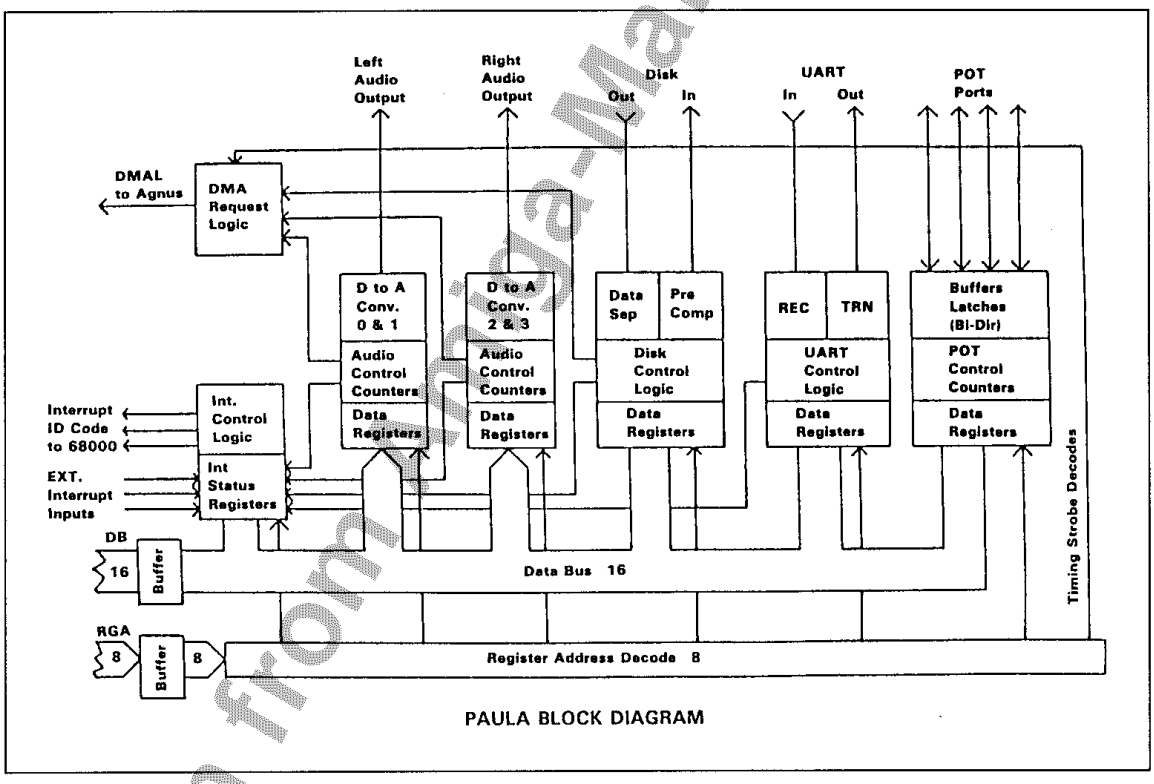


Abb. 2.1: Blockschaltbild Paula

In dem zu einem D/A-Wandler gehörenden Registersatz befindet sich neben den oben erwähnten Registern auch das eigentliche Datenregister des Wandlers, in das die Sampling-Werte entweder durch den Prozessor oder durch die DMA geschrieben werden. Da dieses wie alle Customchip-

Taken from Amiga-Manuals-Website

Register 16 Bit breit ist und die D/A-Wandler nur eine Auflösung von 8 Bit aufweisen, werden von Prozessor oder DMA stets zwei Sampling-Werte gleichzeitig in das Register übertragen, zusammengefaßt zu einem 16-Bit-Datenwort. Bei der Wiedergabe wird nun nach einem bestimmten Zeitintervall, das durch einen internen Zähler bestimmt wird, zuerst das höherwertige Byte an den Wandler ausgegeben. Dann wird der Zähler wieder auf den im zugehörigen Register definierten Startwert gesetzt und erst nach abermaligem Herunterzählen das niederwertige Byte ausgegeben. Anschließend erfolgt, je nach Betriebsart, entweder eine DMA-Anforderung nach einem weiteren Datenwort, oder es wird ein Interrupt ausgelöst, in dem der Prozessor weitere Daten zur Verfügung stellen muß. Werden diese Anforderungen jeweils nicht erfüllt, so geht die Kontrollogik des Audiokanals in den Ruhezustand (Idle State) über. Wird im DMA-Betrieb das Ende der wiederzugebenden Datentabelle erreicht, so wird wieder am Anfang der Tabelle mit der Ausgabe begonnen, was eine einfache Möglichkeit darstellt, konstante Töne zu erzeugen.

Jedes der übergebenen 16-Bit-Datenworte besteht also aus zwei 8 Bit breiten Samples. Die Sampling-Werte selbst müssen in der Zweierkomplement-Form (siehe Kap. 1.2.1.1) dargestellt sein. Der Wertebereich der Wandler erstreckt sich also von 127 – 128. Der Rauschabstand, der sich aus dieser Auflösung ergibt, errechnet sich so:

$$\textbf{Rauschabstand in dB} = 20 * \log 2^{-8} \approx 48 \text{ dB}$$

Formel 25

Daß dies nicht gerade berauschend viel ist, wird bei einem Vergleich zum HiFi-Sektor offenbar: dort werden im Home-Bereich Rauschabstände von mindestens 80 dB, im Profibereich bis 120 dB verlangt. Da hier jedoch Signale mit beliebiger Amplitude verarbeitet werden müssen, ergeben sich bei leisen Passagen naturgemäß wesentlich ungünstigere Signal-Rausch-Verhältnisse. Dieses Problem läßt sich beim Amiga nur durch penible Einhaltung der Regel kompensieren, stets die höchstmögliche Aussteuerung zu wählen und jegliche Dynamik- und Hüllkurvenmodulation mit Hilfe des Lautstärkeregisters vorzunehmen.

Dieses Lautstärkeregister weist eine Breite von 7 Bit auf. Es sind jedoch nur die Werte von 0 bis 64, also 65 verschiedene Lautstärken einstellbar. Dabei ist die Ausgangsspannung zum Wert in diesem Register proportional, d.h., bei einem Wert von 64 ergibt sich die volle Ausgangsspannung, bei 32 die halbe, bei 16 ein Viertel derselben und so fort. Die maximale Ausgangsspannung der Wandler beträgt bei voll aufgedrehtem Lautstärkeregister ungefähr +0.4 Volt für den Eingangswert 127, 0 Volt bei 0 und –0.4 Volt bei einem Eingangswert von –128. Nimmt man diesen Pegel als 0 dB an, so ergeben sich für die verschiedenen Werte im Lautstärkeregister die folgenden dB-Zahlen:

Lautstärkeregister	dB-Wert	Lautstärkeregister	dB-Wert
64	0	32	–6.0
63	–0.1	31	–6.3
62	–0.3	30	–6.6
61	–0.4	29	–6.9
60	–0.6	28	–7.2
59	–0.7	27	–7.5
58	–0.9	26	–7.8
57	–1.0	25	–8.2
56	–1.2	24	–8.5
55	–1.3	23	–8.9
54	–1.5	22	–9.3
53	–1.6	21	–9.7
52	–1.8	20	–10.1
51	–2.0	19	–10.5
50	–2.1	18	–11.0
49	–2.3	17	–11.5
48	–2.5	16	–12.0
47	–2.7	15	–12.6
46	–2.9	14	–13.2
45	–3.1	13	–13.8
44	–3.3	12	–14.5
43	–3.5	11	–15.3
42	–3.7	10	–16.1
41	–3.9	9	–17.0
40	–4.1	8	–18.1
39	–4.3	7	–19.2
38	–4.5	6	–20.6
37	–4.8	5	–22.1
36	–5.0	4	–24.1
35	–5.2	3	–26.6
34	–5.5	2	–30.1
33	–5.8	1	–36.1
		0	– [unendl.]

Tab. 2.1: Lautstärkeregister- und dB-Werte

2.1.2 Agnus – die DMA-Kontrolle

Während Paula bezüglich der Disksteuerung und der Audio-Kanäle immer nur Anforderungen von sich gibt, ist Agnus der fleißige Arbeiter, der alle diese Komponenten von Agnus mit den benötigten Daten versorgt. Wie oben erwähnt, bezeichnet man diesen Vorgang der prozessor-

Taken from Amiga-Manuals-Website

unabhängigen Datenübertragung als DMA. Die DMA-Übertragung muß naturgemäß über dieselben Daten- und Adreßleitungen erfolgen, die auch der Prozessor benutzt. Um diesen dabei möglichst wenig in seiner Arbeit zu behindern, nutzt Agnus vorzugsweise diejenigen Buszyklen, in denen der Prozessor den Bus nicht benötigt; dies sind architekturbedingt die ungeraden Zyklen. Lediglich der Video-DMA und dem Blitter wird das Recht eingeräumt, den Bus auch während der geraden Zyklen zu besetzen und damit den Prozessor zu bremsen.

Das gesamte DMA-Timing orientiert sich praktischerweise am Aufbau einer Video-Rasterzeile, d.h. dem Zeitintervall, in dem der Elektronenstrahl des Monitors einmal von links nach rechts über den Bildschirm gewandert ist und damit eine Zeile des Bildes gezeichnet hat. Innerhalb dieses Intervalls hat die DMA wegen ihrer begrenzten Geschwindigkeit nur eine bestimmte Anzahl von Gelegenheiten zur Datenübertragung zur Verfügung, nämlich vorzugsweise die ungeraden Buszyklen und nur im Ausnahmefall auch die geraden. Für die Hardware-Designer lag es nun nahe, jedem dieser zur DMA-Übertragung nutzbaren Buszyklen (im folgenden DMA-Slots genannt) eine feste Aufgabe zuzuordnen. So dienen die ersten vier DMA-Slots in einer Rasterzeile dem Refresh (Wiederauffrischung) des Chip-RAMs, die folgenden zwei der Disk-DMA, in den nächsten vier werden die einzelnen Audiowandler versorgt, worauf sich die DMA-Slots für die Sprite-DMA anschließen. Erst dann beginnt die sog. Bitplane-DMA, die für den Bildaufbau verantwortlich ist.

Jeder der vier Audiokanäle kann also nur einmal pro Rasterzeile mit Daten versorgt werden. Dies führt natürlich zu einer Beschränkung der maximal möglichen Ausgaberate. Nimmt man eine Bildfrequenz von 50 Bildern pro Sekunde und eine Anzahl von 312.5 Rasterzeilen pro Bild (im PAL-Standard, mit dem alle europäischen Amigas ausgerüstet sind, haben die einzelnen Bilder abwechselnd 312 und 313 Zeilen) und berücksichtigt außerdem, daß bei jedem DMA-Vorgang stets zwei Samples übergeben werden, so erhält man für die höchstmögliche Ausgaberate:

2 Samples/Zeile * 312.5 Zeilen/Bild * 50 Bilder/Sekunde = 31250 Hz

Formel 26

Dies ist jedoch leider nur ein theoretischer Maximalwert. Aus unerfindlichen Gründen (alle dem Autor diesbezüglich gegebenen Auskünfte erwiesen sich als falsch) schafft die Hardware nämlich nur eine Wiedergaberate von höchstens 29504 Samples/Sekunde beim PAL-Amiga, 28867 beim amerikanischen NTSC-Amiga. Wie wir aber im nächsten Abschnitt sehen werden, ist der effektive Nutzbereich des vom Amiga gelieferten Audiosignals normalerweise durch die Filter so begrenzt, daß solch hohe Wiedergaberaten im allgemeinen gar nicht nötig sind.

Wie erwähnt, leitet sich die Wiedergaberate aus dem Zeitintervall ab, das der interne Zähler der Kontrollogik (Periodenzähler) bis zum Erreichen des 0-Wertes benötigt. Dieses Zeitintervall wird natürlich aus dem Startwert bestimmt, mit dem der Zähler zu Beginn geladen wird. Mit jedem

Buszyklus wird der Zähler dann um 1 dekrementiert (heruntergezählt). Da der europäische PAL-Amiga mit 7.14 MHz getaktet ist und ein Buszyklus zwei Taktzyklen in Anspruch nimmt, ergibt sich eine Buszyklus-Frequenz von 3.57 MHz und eine Dauer eines Buszyklus von 0.2801 µs. Möchte man nun zu einer bestimmten Ausgaberate den entsprechenden Zähler-Startwert errechnen, so muß man die Buszyklus-Frequenz durch diese teilen:

$$\text{Periodenzähler-Wert} = \frac{\text{Buszyklus-Frequenz}}{\text{Ausgaberate}}$$

Formel 27

Bei dem oben erwähnten Maximalwert für die Ausgaberate ergibt sich also ein minimal zulässiger Wert des Periodenzählers von:

$$\text{minimaler PZ-Wert} = \frac{3570000\ \text{Hz}}{29504\ \text{Hz}} = 121$$

Formel 28

Wird dieser Wert unterschritten, so kann die DMA in bestimmten Momenten das angeforderte Datenwort nicht mehr rechtzeitig liefern. Dies hat zur Folge, daß nicht die gewünschten, sondern die beiden Samples des zuletzt übergebenen Datenwortes noch einmal ausgegeben werden, was natürlich eine nicht unerhebliche Verzerrung verursachen würde.

Diesen Effekt kann man sich jedoch andererseits auch zunutze machen: Läßt man nämlich die DMA-Logik lediglich ein einziges Datenwort immer wiederholen, so ist es gleichgültig, ob der Datentransfer rechtzeitig stattgefunden hat, da sich durch ihn der Inhalt der beiden auszugebenden Bytes im Datenregister sowieso nicht ändert. Umgeht man die Anti-Aliasing-Filter oder schaltet sie ab und setzt die Lautstärke zusätzlich auf das Maximum, so lassen sich durch entsprechende Wahl des Periodenzähler-Wertes sogar Frequenzen bis 1.79 MHz erzeugen!

2.2 Klangwiedergabe

2.2.1 Die Berechnung von Klangdaten

Kommen wir nun zu den abzuspielenden Daten und deren Klangeigenschaften selbst. Wie wir gesehen haben, kann der Amiga nicht nur einzelne Töne, sondern sogar ganze Musikpassagen, die in digitalisierter Form vorliegen, ohne Eingreifen des Prozessors am Stück wiedergeben. Dies hat jedoch den Nachteil eines immensen Speicherplatzbedarfs: Bei einer Sampling-Rate von 15 kHz

verbrauchen 10 Sekunden Musik bereits 150 Kbyte Speicher, und das gesamte verfügbare Chip-RAM reicht bei dieser Samplingrate für nicht einmal 30 Sekunden. Viel platzsparender ist es hingegen, die Sampling-Fähigkeiten des Amiga dahingehend zu nutzen, daß man mit kleinen Datenfeldern und deren kontinuierlicher Wiederholung konstante Wellenformen erzeugt, die vom Prozessor bezüglich ihrer Frequenz und Lautstärke moduliert und so als »Instrumente« eingesetzt werden können.

Möchte man durch die Konstruktion eines solchen Sample-Datenfeldes einen bestimmten Klang erzielen, so muß man sich vorher ungefähr über die Oberwellenverteilung des gewünschten Klanges im klaren sein. Einige für die elektronische Musik typische Wellenformen und ihre Spektralverteilungen haben wir ja schon in Kap. 1.1.2 kennengelernt. Wir wollen nun untersuchen, wie man mit der Methode der Aufaddierung von Teilschwingungen einen beliebigen Klang konstruieren kann und welche Rahmenbedingungen bezüglich der Ausgaberate dabei erfüllt sein müssen, um eine einwandfreie Wiedergabe zu erhalten.

Wiedergaberate und Samples/Cycle

Zuerst einmal wollen wir bestimmen, aus wie vielen Elementen, also einzelnen Samples unsere Daten bestehen müssen, um den Hardware-Gegebenheiten gerecht zu werden. Dazu betrachten wir zunächst einmal eine reine Sinusschwingung, von der wir eine Tabelle eines einzigen Schwingungsdurchlaufs (engl.: Cycle) anfertigen wollen. Die Anzahl der Samples, die zur Bildung eines solchen Cycles nötig sind, werden wir im folgenden als Samples/Cycle bezeichnen. Die effektive Frequenz, die bei der Wiedergabe aus dem Lautsprecher zu hören ist, errechnet sich aus:

$$f_{eff} = \frac{f_W}{\text{Samples/Cycle}}$$

mit f_{eff} - effektive Frequenz

f_W - Wiedergaberate

Formel 29

Da ja hardwarebedingt keine größere Wiedergaberate f(W) als 28867 Hz möglich ist, gilt für die größte wiedergebbare Frequenz:

$$f_{maxeff} \leq \frac{28867\ \text{Hz}}{\text{Samples/Cycle}}$$

Formel 30

Wenn man also auf jeden Fall Frequenzen bis zu einer bestimmten Grenze f(maxeff) erzeugen will, so darf man keine höhere Anzahl an Samples verwenden als:

$$\text{Samples/cycle} \leq \frac{28867\ \text{Hz}}{f_{maxeff}}$$

Formel 31

Andererseits haben wir im Kapitel »Grundlagen« gesehen, daß die Tonhöhe und damit die Wiedergaberate auch nicht zu tief gewählt werden darf, da sonst Aliasing-Frequenzen unterhalb der Filtergrenze zu liegen kommen und hörbar werden. Damit dies nicht der Fall ist, muß die Bedingung erfüllt sein:

$$\text{fmineff} = \frac{f_W}{\text{Samples/Cycle}} < f_W - f_{Grenz}$$

mit f_{Grenz} - Grenzfrequenz des Filters

Formel 32

Damit also bei einer reinen Sinusschwingung keine Aliasing-Komponenten auftreten, müssen mindestens soviel Samples verwendet werden, wie sich aus der Auflösung der obigen Bedingung ergeben:

$$\text{Samples/Cycle} > \frac{f_W}{f_W - f_{Grenz}}$$

Formel 33

Bei einem gegebenen Wert für Samples/Cycle bedeutet das eine Untergrenze für die Wiedergaberate von *(s. Formel 35)*

$$f_W > f_{Grenz} * \frac{\text{Samples/Cycle}}{\text{Samples/Cycle-1}}$$

Formel 34

und eine tiefste effektive Frequenz von:

$$\text{fmineff} > \frac{f_{Grenz}}{\text{Samples/Cycle-1}}$$

Formel 35

Hierbei kann es vorkommen, daß der ideale Wert für Samples/Cycle genau zwischen zwei geraden Zahlen liegt, und man gezwungen ist, diesen aufzurunden. Hier kann man sich jedoch mit einem Trick behelfen: Wenn wir beispielsweise einen Samples/Cycle-Wert von 3.5 erzielen wollen, brauchen wir nur 14 Samples zugrundezulegen, in denen vier Schwingungszyklen enthalten sind. So läßt sich durch die geeignete Wahl des Verhältnisses zwischen der Anzahl der Samples und der darin enthaltenen Zyklen nahezu jeder Bruch beliebig genau annähern.

Oberwellen

Dies alles gilt wie gesagt nur für den Fall, daß eine reine Sinusschwingung vorliegt, die keine weiteren Oberwellen enthält. Bei allen anderen Wellenformen müssen wir jedoch berücksichtigen, daß außer der Grundschwingung eben noch die entsprechenden Oberwellen in den hörbaren Bereich gespiegelt werden könnten. Dadurch wird der »Einsatzbereich« unserer Wellenform natürlich noch weiter eingeschränkt. Nehmen wir an, die höchste Frequenz f(n) in dieser Wellenform sei die Oberwelle n-ter Ordnung, sie sei also bei einer bestimmten Grundfrequenz f(0) gleich n * f(0). Während die obere, hardwarebedingte Grenze für Wiedergaberate und Grundfrequenz gleich bleibt, müssen wir unseren Ansatz für die untere Grenze folgendermaßen korrigieren:

$$n * \frac{f_W}{\text{Samples/Cycle}} < f_W - f_{Grenz}$$

Formel 36

Dies ergibt für die Mindestanzahl von Samples/Cycle

$$\text{Samples/Cycle} > n * \frac{f_W}{f_W - f_{Grenz}}$$

Formel 37

und für die Untergrenzen von Wiedergaberate und der damit verbundenen kleinsten effektiven (Grund-)Frequenz

$$f_W > f_{Grenz} * \frac{\text{Samples/Cycle}}{\text{Samples/Cycle-n}}$$

$$f_{mineff} > \frac{f_{Grenz}}{\text{Samples/Cycle-n}}$$

Formel 38

Nun haben wir aber in Kap. 1.1.2 gesehen, daß wohl die meisten aller möglichen Wellenformen aus einer unendlichen Anzahl von Oberwellen zusammengesetzt sind. Müßte man diese alle berücksichtigen, so würde der nutzbare Bereich auf 0 zusammenschrumpfen, und es wäre im Prinzip überhaupt keine störungsfreie Wiedergabe der Wellenform möglich. Wie jedoch im Abschnitt über die Filter schon erwähnt, sind Störgeräusche ab einem effektiven Abstand zum Nutzsignal von etwa 30 dB nicht mehr hörbar, je nach Frequenzverhältnis der Nutz- und Störanteile auch bei weniger. Möchte man also den Wert für n bestimmen, d.h. die Ordnung der höchsten zu berücksichtigenden Oberwelle, so muß man sich die Frage stellen, ab welcher Ordnung alle folgenden Oberwellen schwach genug sind, um vernachlässigt werden zu können.

Nehmen wir als Beispiele die in Abb. 1.6 dargestellten Wellenformen: Bei einer Dreieckschwingung hat die Oberwelle n=3 ein Neuntel der Lautstärke der Grundschwingung, n = 5 hat $^1/_{25}$, n=7 hat $^1/_{49}$ und so fort. Da ein Lautstärkenverhältnis von –30 dB ein Spannungsverhältnis von ca. $^1/_{32}$ bedeutet, liegt die Oberwelle der 7. Ordnung schon deutlich unterhalb der Hörschwelle, so daß in den obigen Ungleichungen für n eine 5 eingesetzt werden kann. Nimmt man dagegen eine Sägezahnschwingung, so ergibt sich für n ein Wert von 32, da diese Wellenform über einen wesentlich größeren Gehalt an Oberwellen verfügt. Lediglich das Rechteck nimmt hier eine Sonderstellung ein: Seine Oberwellen sind ja genau mit ihren eigenen Aliasing-Freqenzen

identisch, so daß man für die Wiedergabe einer Rechteckschwingung nur zwei Samples benötigt oder, falls man verschiedene Tastverhältnisse erhalten möchte, genau so viele Samples wie die Anzahl der Schritte, in denen das Tastverhältnis variiert werden soll.

In diesem Zusammenhang soll noch einmal deutlich auf den folgenden Punkt hingewiesen werden: Sämtliche in diesem Abschnitt angeführten Formeln und Ungleichungen für die minimale Wiedergaberate oder die Mindestanzahl der Samples/Cycle ergeben lediglich Grenzwerte. Sie machen keine Aussage über die effektive Hörbarkeit von eventuell auftretenden Spiegelungs-Frequenzen bzw. darüber, wie störend sich diese wirklich auswirken. Es wurde hier außerdem vernachlässigt, daß der verwendbare Bereich einer Wellenform ja auch durch den Filter selbst eingeschränkt wird. Man sollte deshalb bei einem gegebenen Frequenzbereich, den man mit seiner Wellenform abdecken möchte, bei der Wahl der Oberwellen auch berücksichtigen, wieviel davon überhaupt vom Filter durchgelassen wird. Des weiteren sollte man nicht vergessen, daß die Begrenzungseigenschaften des Filters keineswegs ideal sind, da vom Beginn des Sperrbereichs bis zur Dämpfung auf –30dB ein Bereich von gut 3 kHz verbleibt. Dies hat unter anderem zur Folge, daß z.B. bei der Beurteilung der Hörbarkeit von Oberwellen und deren Aliasingfrequenzen die Grenzwerte nicht immer so streng angesetzt werden müssen.

Nachdem wir den Wert für Samples/Cycle ermittelt haben, können wir nun an die eigentliche Konstruktion unserer Wellenform gehen. Hierzu müssen wir zuerst eine Folge reeller Zahlen aufstellen, die den Verlauf der Schwingung im Verhältnis wiedergibt. Diese Folge gewinnt man nach der Formel:

$$r_t = \sum_{k=1}^{n} \left(a_k * \sin\left(k * 2\pi * \frac{t}{\text{Samples/Cycle}}\right) \right)$$

mit

- t - Nummer des Samples in der Tabelle
- k - Index der einzelnen Oberwellen
- n - Ordnung der höchsten Oberwelle
- a_k - Fourier-Transformation der k-ten Oberwelle

Formel 39

Im nächsten Schritt müssen die Werte dieser Folge noch an den Darstellungsbereich des D/A-Wandlers angepaßt werden. Hierbei muß man jeweils das Maximum und das Minimum der Folge ermitteln, also die Stelle mit dem kleinsten und die mit dem größten Wert in der Folge. Wenn Sie diese nach der obigen Formel nur aus Sinuskomponenten berechnet haben, werden beide dem Betrag nach identisch sein, so daß sie die Wertetabelle der Wellenform so berechnen können *(wie in Formel 41 dargestellt)*:

$$s_t = \mathrm{rnd}\left(r_t * \frac{128}{r_{max}} \right)$$

mit rnd() - Rundungsfunktion

r_{max} - Wertemaximum

Formel 40

Sollten Sie jedoch noch andere als Sinuskomponenten bei der Berechnung verwendet haben, so müssen Maximum und Minimum nicht mehr notwendig gleich sein. Dann empfiehlt sich die Verwendung der Formel:

$$s_t = \mathrm{rnd}\left(r_t * \frac{256}{r_{max}-r_{min}} - \frac{r_{max}-r_{min}}{2} \right)$$

Formel 41

Wenn sie beliebige Wellenformen zu den Sinusanteilen hinzufügen, sollten sie außerdem darauf achten, daß diese mit demselben Wert beginnen und enden. Andernfalls tritt bei der Wiederholung der Tabellenwerte ein Spannungssprung auf, der ebenfalls unerwünschte Oberwellen, etwa von der Intensität eines entsprechenden Sägezahns, erzeugt.

2.2.2 DMA-Betrieb

Die Register der Custom-Chips des Amiga liegen alle nebeneinander im Adreßbereich $DFF000 bis $DFF280. Sie sind durchgehend 16 Bit breit, weshalb alle Register, die (32-Bit-)Adressen beherbergen müssen, zu jeweils einem höherwertigen (High-) und einem niederwertigen (Low-)Teil zusammengefaßt sind. Da diese beiden Teile stets direkt beieinanderliegen, können mittels einer Langwortübertragung auch beide Datenwörter zugleich in das Registerpaar übertragen werden. Die für die Audiohardware zuständigen Register liegen im Bereich ab $DFF0A0. Für jeden Audiokanal sind sechs Register vorgesehen, wobei die Register für Kanal 0 ab $DFF0A0, die für Kanal 1 ab $DFF0B0, für Kanal 2 ab $DFF0C0 und für Kanal 3 ab $DFF0D0 liegen. Wie erwähnt, sind die Kanäle 0 und 3 auf den linken, die Kanäle 1 und 2 auf den rechten Stereoausgang des Amiga geschaltet.

Die AUDxLC-Register

Im DMA-Betrieb müssen wir der Audio-DMA-Kontrolle von Paula zuerst einmal mitteilen, woher die Wiedergabedaten geholt werden sollen. Zu diesem Zweck dient das Registerpaar

AUDxLCH und *AUDxLCL*, wobei das »x« für die Nummer des Audiokanals, 0,1,2 oder 3 steht. In ersterem wird das höherwertige, in letzterem das niederwertige Wort der Adresse gespeichert, ab der die DMA mit der Übertragung beginnen soll. Der Wert dieser beiden Register stellt jedoch keineswegs die aktuelle Adresse dar, an der die DMA im Verlauf der Ausgabe angelangt ist. Der hier angegebene Wert wird vielmehr nur beim Start der DMA und dann jedesmal bei Erreichen des Tabellenendes in den eigentlichen Zeiger übertragen, der die Adresse des nächsten zu übertragenden Wortes enthält. Auf diesen Zeiger hat der User keinen Zugriff, weshalb es u. a. nicht möglich ist, einen einmal für längere Zeit unterbrochenen DMA-Vorgang an der gleichen Stelle fortzusetzen.

Die Register *AUDxLCH* und *AUDxLCL* belegen innerhalb des Chip-Registerbereiches die folgenden Adressen:

$DFF0A0 [AUD0LCH] Audiodatenzeiger Kanal 0 Bit 16–18
$DFF0A2 [AUD0LCL] Bit 0–15
$DFF0B0 [AUD1LCH] Audiodatenzeiger Kanal 1 Bit 16–18
$DFF0B2 [AUD1LCL] Bit 0–15
$DFF0C0 [AUD2LCH] Audiodatenzeiger Kanal 2 Bit 16–18
$DFF0C2 [AUD2LCL] Bit 0–15
$DFF0D0 [AUD3LCH] Audiodatenzeiger Kanal 3 Bit 16–18
$DFF0D2 [AUD3LCL] Bit 0–15

Wie man auch hier sieht, haben alle Adressen, von denen die DMA Daten übertragen kann, lediglich eine Breite von 18 Bit. Das ergibt sich aus der Tatsache, daß der Adreßbus der DMA-Vorrichtung im Agnus-Chip leider nur 18 Leitungen hat und so nur die untersten 512 Kbyte des Amiga-Systems, das sog. Chip-RAM, per DMA erreicht werden können. Der neue BigAgnus-Chip (8372A), der bereits in den neueren Amiga 500 und Amiga 2000 Modellen ab Werk eingebaut ist, erlaubt die DMA-Ansteuerung von 1 Mbyte Chip-Memory. Unter der neuen Betriebssystem-Version Kickstart 2.0, eingebaut im Amiga 3000, ist sogar die Ansteuerung von 2 Mbyte Speicher möglich!

Nehmen wir nun also an, wir wollten über Kanal 0 eine aus 16 Samples bestehende Sinus-Wellenform als unveränderlichen, konstanten Ton wiedergeben. Hierzu berechnen wir uns zuerst eine entsprechende Tabelle:

```
sampledata:
        dc.b     0,   49
        dc.b    90,  117
        dc.b   127,  117
        dc.b    90,   49
        dc.b     0,  -49
        dc.b   -90, -117
        dc.b  -127, -117
        dc.b   -90,  -49

sdend:

sdlen   EQU   sdend-sampledata      ; Länge der Daten in Bytes
```

Taken from Amiga-Manuals-Website

Dann übertragen wir die Startadresse dieser Daten in *AUD0LC*. In 68000- Assembler würde das so aussehen:

```
   AUD0LCH EQU $DFF0A0
   AUD0LC  EQU AUD0LCH

SetCH0DataLoc:

   lea     sampledata,a0
   move.l  a0,AUD0LC
```

Die AUDxLEN-Register

Um nun einen bestimmten Speicherbereich zu definieren, müssen wir der Hardware als nächstes die Länge des abzuspielenden Speicherbereiches mitteilen. Dafür ist das Register *AUDxLEN* vorgesehen. Hier muß der Prozessor die Tabellenlänge in Wörtern bzw. in Bytes durch 2 angeben. Da in diesem Register die volle Bitbreite ausgeschöpft werden kann, beträgt der größte Speicherbereich, der auf diese Weise erfaßt werden kann, 216+1 Byte oder 128 Kbyte. Die Adressen für die einzelnen Kanäle lauten:

$DFF0A4 [AUD0LEN] Tabellenlänge in Wörtern Kanal 0
$DFF0B4 [AUD1LEN] Tabellenlänge in Wörtern Kanal 1
$DFF0C4 [AUD2LEN] Tabellenlänge in Wörtern Kanal 2
$DFF0D4 [AUD3LEN] Tabellenlänge in Wörtern Kanal 3

Wie bereits im Abschnitt über Paula erwähnt wurde, ist die Hardware dafür ausgelegt, eine einmal gewählte Audio-Datentabelle ständig zu wiederholen, so lange der Prozessor nicht eingreift und eine neue Einstellung vornimmt. Dies wird von Paula folgendermaßen bewerkstelligt: Beim Start des DMA-Vorgangs werden sowohl die Startadresse in *AUDxLCH* und *AUDxLCL* in den internen Zeiger übertragen als auch die Länge der Tabelle aus dem *AUDxLEN*-Register in einen ebenfalls internen Zähler. Nun wird bei jedem erfolgten DMA-Zugriff der Datenzeiger auf das nächste zu übertragende Wort gestellt und der interne Zähler um eins dekrementiert. Das wird so lange fortgesetzt, bis der Zähler bei 0 angelangt ist. Dann werden Zeiger und Zähler wieder auf die in den Registern angegebenen Werte gesetzt und der Vorgang beginnt somit von neuem.

Hieraus wird unter anderem deutlich, wie beliebige Audiodaten bündig hintereinander abgespielt werden können: Da die Registerwerte nur beim Start der DMA und beim Erreichen des Tabellenendes übernommen werden, eine zwischenzeitliche Änderung der Registerinhalte aber keine Wirkung auf den momentanen Zustand der DMA hat, braucht man nur nach einer Wertübernahme die Register mit den neuen Werten zu versehen, damit nach Beendigung der gerade abgespielten Wellenform sofort die Ausgabe der nächsten beginnt. Wie dies im einzelnen geschieht, werden wir später bei der Interrupt-Handhabung sehen.

Lassen wir also dem *AUDxLEN*-Register die benötigten Daten zukommen. Die Länge unserer Tabelle beträgt 16 Byte, was in dem Label *sdlen* vermerkt ist. Dieser Wert muß noch durch 2 geteilt werden, um die Anzahl der Datenworte zu erhalten. Der entsprechende Assemblerbefehl lautet:

Taken from Amiga-Manuals-Website

```
   AUD0LEN   EQU   $DFF0A4

SetCh0DataLen:

   move.w    #sdlen/2,AUD0LEN
```

Die AUDxVOL-Register

Nun sollten wir außerdem die Lautstärke unseres Tones bestimmen. Für diesen Parameter ist das Register *[AUDxVOL]* vorgesehen. Wie im Abschnitt 2.1.1 beschrieben, sind in diesem Register 7 Bit von Interesse, der zulässige Wertebereich erstreckt sich jedoch nur von 0 bis einschließlich 64, also von binär %00000000 bis %01000000, wobei 0 für gar kein Signal ([–unendl.] dB) und 64 für volle Lautstärke steht (0 dB). Eine genaue Aufschlüsselung der dB-Werte findet man in Tab. 2.1. Die *AUDxVOL*-Register belegen folgende Adressen:

$DFF0A8 [AUD0VOL] Lautstärke Kanal 0
$DFF0B8 [AUD1VOL] Lautstärke Kanal 1
$DFF0C8 [AUD2VOL] Lautstärke Kanal 2
$DFF0D8 [AUD3VOL] Lautstärke Kanal 3

Nehmen wir für unseren Ton der besseren Hörbarkeit halber die höchste Lautstärke:

```
   AUD0VOL   EQU   $DFF0A8

SetCh0Vol:

   move.w    #64,AUD0VOL
```

Die AUDxPER-Register

Als letzter Parameter muß noch die Frequenz des abzuspielenden Tones angegeben werden. Wie schon im Abschnitt 2.1.1 beschrieben, ergibt sich die Wiedergaberate und damit die effektive Frequenz des Tones aus dem Wert eines Zählers, der nach der Ausgabe eines jeden Samples wieder mit diesem Startwert geladen wird und dann mit jedem Clockzyklus um 1 dekrementiert wird. Erst wenn dieser bei 0 angelangt ist, wird das nächste Sample an den Wandler übergeben. Aus diesem Grund wird die Wiedergaberate nicht direkt in Hertz, sondern in der Anzahl der Buszyklen angegeben, die zwischen den Ausgabezeitpunkten zweier Samples liegen. Da die für die Erzeugung einer bestimmten Frequenz f(0) notwendige Wiedergaberate f(W) sich so errechnet.

$$f_W = f_0 * \text{Samples/Cycle}$$

Formel 42

Der Wert für den Periodenzähler läßt sich angeben mit

$$\text{Periode} = \frac{\text{Clock-Frequenz}}{f_W}$$

Formel 43

Man kann den Zählerwert, der für die Wiedergabe der gewünschten Tonhöhe vonnöten ist, mit folgender Formel erhalten:

$$\text{Periode} = \frac{\text{Clock-Frequenz}}{f_0 * \text{Samples/Cycle}}$$

Formel 44

Das Register, mit dessen Wert der interne Zeitzähler nach jedem Ausgabevorgang aufgefrischt wird, heißt *AUDxPER*. Wie alle Customchip-Register verfügt es über eine Breite von 16 Bit, also einen Wertebereich von 0 bis 65535. Die *AUDxPER*-Register der einzelnen Kanäle liegen an den Adressen:

$DFF0A6 [AUD0PER] Periodenzähler Kanal 0
$DFF0B6 [AUD1PER] Periodenzähler Kanal 1
$DFF0C6 [AUD2PER] Periodenzähler Kanal 2
$DFF0D6 [AUD3PER] Periodenzähler Kanal 3

Wie jedoch im Abschnitt 2.1.2 (Agnus) erwähnt, sind der Wiedergaberate gewisse hardwarebedingte Grenzen gesetzt. Die theoretische Obergrenze bildet die Anzahl der Videozeilen, die in einer Sekunde auf den Bildschirm gebracht werden, da jeder Audiokanal nur einmal pro Videozeile mit je zwei Samples versorgt werden kann. Die tatsächlichen Obergrenzen (28867 NTSC und 29504 PAL) ergeben für den Periodenzähler einen minimalen Wert von 124 für NTSC und 121 für den PAL-Amiga, der in Europa verkauft wird. Wird dieser Wert unterschritten, so kann die DMA das nächste Datenwort zuweilen nicht rechtzeitig zur Verfügung stellen, so daß die Wellenform verfremdet wird. Nur wenn im *AUDxLEN*-Register der Wert 1 steht und damit immer dasselbe Datenwort ausgegeben wird, macht es keinen Unterschied, ob dieses rechtzeitig übertragen wird oder nicht. Dies läßt bei Ausschaltung des Audio-Filters (siehe 2.1.3) die Erzeugung einer Rechteckschwingung bis zu einem Viertel der Busfrequenz (1.785 MHz) zu.

Nehmen wir an, wir wollten unsere Wellenform mit einer effektiven Frequenz von genau 1000 Hz wiedergeben. Die dazu nötige Ausgaberate *(in Formel 46)*

$$f_W = 1000\ \text{Hz} * 16\ \text{Samples/Cycle} = 16000\ \text{Hz}$$

Formel 45

ist für die Hardware nicht zu hoch, und es sind auch keine hörbaren Aliasing-Frequenzen zu erwarten. Also errechnen wir den zugehörigen Periodenwert:

$$\text{Periode} = \frac{\text{Clock-Frequenz}}{f_W} = \frac{3570000\ \text{Hz}}{16000\ \text{Hz}} = 223$$

Formel 46

Dieser Wert muß jetzt nur noch an das *AUDxPER*-Register übergeben werden, damit alles für den Start der Tonausgabe vorbereitet ist:

```
   AUD0PER  EQU  $DFF0A6

SetCh0Per:

   move.w   #223,AUD0PER
```

Da das *AUDxPER*-Register wie alle Hardwareregister nur ganze Zahlen aufnehmen kann, ist man bei der Berechnung der Periodenzählerwertes gezwungen, eine Rundung vorzunehmen. Diese Rundung erzeugt naturgemäß einen gewissen Fehler, d.h., die tatsächlich wiedergegebene Frequenz unterscheidet sich von der gewünschten um einen gewissen Betrag. Dieser kann jedoch nicht größer sein als ein Abstufungsschritt des Periodenzählers im jeweiligen Bereich, in diesem Fall also nicht größer als ca. ein $^1/_{223}$. Dies ist für den Audiobereich normalerweise noch hinzunehmen, für technische Anwendungen jedoch (z.B. Erzeugung und Modulation von Trägerfrequenzen) wird hier eine genaue Fehlerrechnung notwendig. Die Differenz zwischen der gewünschten Frequenz f(0) und der nächsten erreichbaren errechnet sich aus:

$$\Delta f = f_0 - \frac{\text{Clock-Frequenz}}{\text{Samples/Cycle} * \text{Periode}}$$

Formel 47

Da die Periode durch die Rundung des Ausdrucks

$$\text{Periode} = \text{rnd}\left(\frac{\text{Clock-Frequenz}}{\text{Samples/Cycle} * f_0}\right)$$

mit **rnd()** - **Rundungsfunktion**

Formel 48

errechnet wird, ergibt sich durch Einsetzen dieses Ausdrucks in die obige Gleichung:

$$\Delta f = f_0 - \frac{\text{Clock-Frequenz}}{\text{Samples/Cycle} * \text{rnd}\left(\frac{\text{Clock-Frequenz}}{\text{Samples/Cycle} * f_0}\right)}$$

Formel 49

Isoliert betrachtet können diese Differenzen vom menschlichen Ohr so gut wie nicht wahrgenommen werden. Fügt man jedoch mehrere Töne zu einem Akkord zusammen, so können sich durch die Interferenzen zwischen den einzelnen Tönen (Schwebungen) bereits sehr feine Frequenzunterschiede bemerkbar machen, die das Klangbild nicht unerheblich verfälschen und den Akkord »schräg« klingen lassen. Um dies zu vermeiden, hilft in manchen Fällen nur eine Korrektur (d.h. Erhöhung) des Wertes für Samples/Cycle und damit der Sampling-Rate.

In der folgenden Tabelle sind die Periodenzählerwerte für die sogenannte »wohltemperierte Tonleiter« bei der von uns verwendeten 16-Bit-Wellenform angegeben. In der wohltemperierten Tonleiter sind die Frequenzverhältnisse zwischen den einzelnen Tönen genau gleich (im Gegensatz zu den natürlichen Oberwellenschwingungen oder zu der Tonleiter, die man aus der Aneinanderreihung von Quinten gewinnt). Für diese Tonleiter gilt die Bedingung, daß jeder Ton im genau gleichen Frequenzverhältnis zu seinem Nachfolger steht wie der vorhergehende zu jenem, alle Töne also genau den gleichen tonalen Abstand voneinander haben. Hierbei gilt nach wie vor, daß zwei Töne im Abstand von einer Oktave genau im Frequenzverhältnis 1:2 stehen müssen. Da eine Oktave zwölf Töne enthält, muß ein Faktor gefunden werden, der zwölfmal mit sich selbst multipliziert 2 ergibt. Dies ist genau die 12. Wurzel aus 2, oder anders geschrieben $2^{1/12}$ (dezimal 1.059463). Nach diesem Faktor berechnet, ergeben sich bei 16 Samples/Cycle (*AUDxLEN* = 8) die folgenden Periodezählerwerte:

Note	Periode	Ideale Frequenz	Reale Frequenz
A	508	440.0	440.4
A#	480	466.2	466.1
H	453	493.9	493.9
C	428	523.3	522.7
C#	404	554.4	553.8
D	381	587.3	587.2
D#	360	622.3	621.4
E	339	659.3	659.9
F	320	698.5	699.1
F#	302	740.0	740.8
G	285	784.0	785.0
G#	269	830.6	831.7
A	254	880.0	880.8
A#	240	932.3	932.2
H	226	987.8	989.9
C	214	1046.5	1045.4
C#	202	1108.7	1107.5
D	190	1147.7	1177.5
D#	180	1244.5	1242.9
E	170	1318.5	1316.0
F	160	1396.9	1398.3
F#	151	1480.0	1481.6
G	143	1568.0	1564.5
G#	135	1661.2	1657.2

Tab. 2.2: Periodenwerte für die wohltemperierte Tonleiter

Die DMA-Kontrolle

Nachdem wir alle Parameter für unseren Ton festgelegt haben, können wir nun endlich die DMA-Übertragung starten und den Ton erklingen lassen. Für die Kontrolle der einzelnen DMA-Prozesse dient das sog. *DMACON*-Register. In ihm können alle DMA-Sorten wie Bitplane-, Blitter-, Copper-, Sprite- und Audio-DMA durch Setzen oder Löschen der entsprechenden Bits an- oder abgeschaltet werden. Wenn man das zu einem Audiokanal zugehörige DMA-Bit setzt, so werden wie erwähnt die Werte von *AUDxLC*, *AUDxLEN* und *AUDxVOL* in die zugehörigen Register übernommen und die DMA-Übergabe gestartet. Ab jetzt wird die im *AUDxLC*- und *AUDxLEN*-Register definierte Wellenform in ständiger Folge wiederholt, bis das entsprechende Bit im *DMACON*-Register wieder abgeschaltet wird.

Das *DMACON*-Register besitzt als höchstwertiges Bit einen Schalter, der angibt, ob die auf 1 gesetzten Bits des in das Register geschriebenen Wertes die entsprechenden Registerbits auf 0 oder

Talk from Amiga-Manuals-Website

auf 1 setzen sollen. Gibt man also im höchstwertigen Bit eine 1 an, so werden alle sonstigen im Schreibwert auf 1 gesetzten Bits ebenfalls auf 1 gesetzt, bei einer 0 im höchstwertigen Bit werden alle 1-Bits gelöscht, also auf 0 gesetzt. Dies ist notwendig, damit der neue Registerwert nicht erst aus dem vorherigen konstruiert werden muß, da man ja meist nur einzelne Bits verändern will und die restlichen unberührt bleiben sollen. Die für uns interessanten Bits des *DMACON*-Registers sehen folgendermaßen aus:

DMACON-Register

Bit	*Name*	*Funktion*
15	*SETCLR*	*1 = alle 1-Bits werden auf 1 gesetzt, alle anderen bleiben unberührt 0 = alle 1-Bits werden auf 0 gesetzt, alle anderen bleiben unberührt*
9	*DMAEN*	*Schalter für alle DMA-Prozesse*
3	*AUD3EN*	*Audio-DMA Kanal 3 einschalten*
2	*AUD2EN*	*Audio-DMA Kanal 2 einschalten*
1	*AUD1EN*	*Audio-DMA Kanal 1 einschalten*
0	*AUD0EN*	*Audio-DMA Kanal 0 einschalten*

Um den von uns gewählten Audiokanal zur Wiedergabe zu veranlassen, müssen wir das entsprechende Bit im *DMACON*-Register auf 1 setzen, wobei wir nicht vergessen sollten, vorsichtshalber auch das *DMAEN*-Bit mitzusetzen:

```
   DMACONW  EQU  $DFF096  ; DMACON schreiben

   SETCLR   EQU  $0800
   DMAEN    EQU  $0200
   AUD0EN   EQU  $0001

StartCh0DMA:
   move.w  #SETCLR!DMAEN!AUD0EN,DMACONW
```

Jetzt ist die Audio-DMA eingeschaltet und unser Ton erklingt aus dem Lautsprecher. Um die Tonausgabe wieder anzuhalten, müssen wir lediglich eine 0 in das entsprechende Bit des *DMACON*-Registers zu schreiben. Das geschieht so:

```
StopCh0DMA:
   move.w  #AUD0EN,DMACONW
```

Beim nächsten Ausgabezyklus wird also keine DMA-Anforderung mehr gestellt, womit die Kontrollogik in den Ruhezustand übergeht. Der Wert des internen Datenzeigers ist damit natürlich verloren, da er beim nächsten Einschalten der DMA unweigerlich überschrieben wird. Nur wenn der Zyklus, während dessen die DMA abgeschaltet wurde, noch nicht vorüber und DMA-Kontrolle noch nicht im Idle State ist, kann mit der Tonausgabe an der alten Stelle fortgefahren werden.

Taken from Amiga-Manuals-Website

Beispiele

Fassen wir die Schritte, die für die Vorbereitung der Tonausgabe notwendig waren, noch einmal zusammen:

1. Konstruieren Sie Ihre Wellenform (siehe Kap. 2.2.1).
2. Sorgen Sie dafür, daß Ihre Wellenform an einer geraden Adresse im Chip-RAM zu liegen kommt und eine gerade Anzahl von Samples enthält.
3. Setzen Sie die Register für den Zeiger auf die Audiodaten: *AUDxLCH* (Bits 16–18)/ *AUDxLCL* (Bits 0–15)
4. Geben Sie die Länge der Audiodaten in Wörtern im *AUDxLEN*-Register an.
5. Setzen Sie das Lautstärkeregister *AUDxVOL* auf die gewünschte Lautstärke.
6. Definieren Sie die gewünschte Wiedergaberate durch Angabe des Periodenzählerwertes im *AUDxPER*-Register.
7. Starten Sie die Audio-DMA, indem Sie das *SETCLR*-Bit zusammen mit dem *DMAEN*-Bit und dem entsprechenden Bit des gewünschten Audiokanals in das *DMACON*-Register schreiben.

Das folgende Beispielprogramm soll die bisher besprochenen Verfahrensweisen zusammenfassen und verdeutlichen. Stören Sie sich nicht daran, daß die Register- und Konstantennamen, die in diesem und den folgenden Beispielprogrammen verwendet werden, von den oben gezeigten geringfügig abweichen. Dies liegt daran, daß die Definitionen in den offiziellen sog. Include-Files für den Amiga (Dateien mit Konstantendefinitionen, die beim Kompilieren bzw. Assemblieren eingebunden werden) so festgelegt wurden. So lauten beispielsweise die Defininitionen der für den Audio-Bereich zuständigen Register:

```
dmacon  EQU   $096
clxcon  EQU   $098
intena  EQU   $09A
intreq  EQU   $09C
adkcon  EQU   $09E

aud     EQU   $0A0
aud0    EQU   $0A0
aud1    EQU   $0B0
aud2    EQU   $0C0
aud3    EQU   $0D0

 ; STRUCTURE AudChannel,0

ac_ptr      EQU   $00
ac_len      EQU   $04
ac_per      EQU   $06
ac_vol      EQU   $08
ac_dat      EQU   $0A
ac_SIZEOF   EQU   $10
```

Taken from Amiga-Manuals-Website

So wird das Laden der Audioregister, hier z.B. des [AUDxLC]-Registers normalerweise so programmiert:

```
lea     $DFF000,a0
move.l  #sampledata,aud0+ac_ptr
```

Sie sollten bei der Programmierung des Amiga stets diese Include-Dateien verwenden, damit Sie erstens die von Ihnen benötigten Definitionen nicht erst mühsam mit der Hand eintippen müssen und zweitens die Fehlerwahrscheinlichkeit verringert wird.

```
; sinuston.asm


INCLUDE   'exec/exec_lib.i'
INCLUDE   'exec/memory.i'
INCLUDE   'hardware/custom.i'
INCLUDE   'hardware/dmabits.i'
INCLUDE   'hardware/intbits.i'
INCLUDE   'asmsupp.i'

CUSTOM_ADR   EQU   $DFF000

        move.l    4.w,a6
        move.l    #sdlen,d0
        moveq     #MEMF_PUBLIC!MEMF_CHIP,d1
        CALLSYS   AllocMem
        move.l    d0,buffer
        beq.s     End

        lea       sampledata(PC),a0
        movea.l   d0,a1
        moveq     #(sdlen/2)-1,d1

CopyBuf
        move.w    (a0)+,(a1)+
        dbra      d1,CopyBuf

        lea       CUSTOM_ADR,a2
        move.l    buffer(PC),ac_ptr+aud0(a2)       ; Puffer 1 zuerst benutzt
        move.w    #(sdlen/2),ac_len+aud0(a2)       ; Pufferlänge in Wörtern
        move.w    #223,ac_per+aud0(a2)             ; Wiedergaberate  1000 Hz
        move.w    #64,ac_vol+aud0(a2)              ; volle Lautstärke
        move.w    #DMAF_SETCLR!DMAF_AUD0,dmacon(a2)    ; DMA einschalten

WaitL1
        btst      #6,$bfe001                        ; auf linke Maustaste warten
        bne.s     WaitL1

        move.w    #DMAF_AUD0,dmacon(a2)            ; DMA ausschalten

        move.l    buffer(PC),a1                    ; Puffer zurückgeben
        move.l    #sdlen,d0
```

Taken from Amiga-Manuals-Website

```
        CALLSYS     FreeMem

End:    rts

sampledata:
        dc.b        0,49
        dc.b        90,117
        dc.b        127,117
        dc.b        90,49
        dc.b        0,-49
        dc.b        -90,-117
        dc.b        -127,-117
        dc.b        -90,-49

sdend:
        sdlen    EQU   sdend-sampledata          ; Länge der Daten in Bytes

buffer
        ds.l        1
```

2.2.3 Interrupts

Im letzten Abschnitt haben wir die Programmierung der Audio-Hardware soweit kennengelernt, daß wir einen konstanten Ton erzeugen konnten. Außer der Option, die DMA auf andere Daten zu richten oder ganz abzuschalten, war jedoch bisher keine sinnvolle Beeinflussung der DMA-Ausgabe machbar, da man ja keine Information über den aktuellen Stand der Audioausgabe gewinnen konnte und damit die Synchronisation eines Programmes mit der DMA unmöglich war. Es gibt jedoch eine Möglichkeit, einem Programm mitzuteilen, wann die DMA-Kontrolle den Wiedergabevorgang gestartet bzw. den Datenzeiger wieder auf den Anfang zurückgesetzt hat. Der M68000-Prozessor verfügt nämlich über die Fähigkeit zur Unterbrechung eines jeden Programmablaufs: Drei Leitungen, die sogenannten Interrupt-Leitungen, können den Prozessor zum Sprung in bestimmte Routinen veranlassen, deren Startadressen aus dem Bereich von $64 – $78 geholt werden. Die durch die 3 Bits der Interrupt-Leitungen darstellbaren sieben Interrupt-Levels (0 = kein Interrupt, Usermode) werden im Paula-Chip noch einmal auf 14 mögliche Interrupt-Quellen verteilt.

Eines der Interrupt-Levels, nämlich Level 4, wird von den Audio-Interrupts der vier einzelnen Kanäle belegt. Dies läßt sich folgendermaßen einsetzen: Erlaubt man einen Audio-Interrupt, so wird dieser jedesmal genau dann ausgelöst, wenn die Werte aus den *AUDxLC* und *AUDxLEN*-Registern jeweils in die internen Zeiger und Zähler übertragen worden sind. Innerhalb der jetzt angesprungenen Interrupt-Routine können nun die neuen Werte für *AUDxLC* und *AUDxLEN* angegeben werden, damit sich die neue Wellenform nach Beendigung der momentan Abgespielten bündig anschließt.

Für die Kontrolle der Erlaubnis als auch des wirklichen Auftretens von Interrupts gibt es zwei Register: Das *INTENA*- und das *INTREQ*-Register. Beide besitzen sowohl ein Lese- (Read only – »R«) als auch ein Schreibregister (Write only – »W«). Diese haben jeweils die Adressen:

Takeshi's Amiga Manuals-Website

$DFF09A [INTENA] *(schreiben)*
$DFF01C [INTENAR] *(lesen)*

$DFF09C [INTREQ] *(schreiben)*
$DFF01E [INTREQR] *(lesen)*

Im *INTENA*-Register können die möglichen Interrupt-Quellen an- bzw. abgeschaltet werden. Dabei ist jedem Bit eine der möglichen Interrupt-Anlässe zugeordnet. Diese Bits sind sowohl im *INTENA*- als auch im *INTREQ*-Register folgendermaßen angeordnet:

INTENA-Register

Bit	Name	IL	Funktion
15	SETCLR		1= alle 1-Bits werden auf 1 gesetzt, alle anderen bleiben unberührt 0 = alle 1-Bits werden auf 0 gesetzt, alle anderen bleiben unberührt
14	INTEN	(6)	Interrupts erlauben
13	EXTER	6	Interrupt von CIA-B oder Expansion-Port
12	DSKSYN	5	Disk-Synchronisation
11	RBF	5	Eingabepuffer serieller Port voll
10	AUD3	4	Audioregister Kanal 3 gefüllt
9	AUD2	4	Audioregister Kanal 2 gefüllt
8	AUD1	4	Audioregister Kanal 1 gefüllt
7	AUD0	4	Audioregister Kanal 0 gefüllt
6	BLIT	3	Blitter fertig
5	VERTB	3	Beginn vertikale Austastlücke
4	COPER	3	Copper
3	PORTS	2	CIA-A oder Expansion Port IRQ
2	SOFT	1	Soft-Interrupts
1	DSKBLK	1	Disk-DMA beendet
0	TBE	1	Ausgabepuffer serieller Port leer

Möchte man z.B den Audio-Interrupt von Kanal 0 aktivieren, so kann das so geschehen:

```
lea     $dff00,a0
move.w  #INTF_SETCLR!INTF_AUD0,intena(a0)
```

Das *INTREQ*-Register ist dazu da, um die effektiv aufgetretenen Interrupts identifizieren zu können. Hat man mehrere Audio-Interrupts erlaubt, so kann man den Kanal, der den Interrupt verursacht hat, auf folgende Weise bestimmen:

```
lea     $dff000,a0
move.w  intreq(a0),d0
btst    #INTB_AUD3,d0
bne     serveaud3
```

Taken from Amiga-Manuals-Website

```
    btst    #INTB_AUD2,d0
    bne     serveaud2

    btst    #INTB_AUD1,d0
    bne     serveaud1

    btst    #INTB_AUD0,d0
    bne     serveaud0
```

Das folgende Beispielprogramm verwendet die Audio-Interrupts dazu, einen Speicherbereich von beliebiger Länge (darf auch im Fast-RAM liegen) bei einer bestimmten Wiedergaberate abzuspielen. Die dafür benötigten Daten werden aus dem File geholt, dessen Name als Parameter in der Kommandozeile des CLI (!) angegeben wird. Hierzu werden zuerst zwei Puffer angefordert, deren Länge in der Konstanten *BUFLEN* festgelegt wird. Dann wird der erste Puffer mit Daten gefüllt und dessen Wiedergabe angeordnet, wobei der Interrupt für den entsprechenden Kanal erlaubt wird. In diesem Interrupt wird dann, während der eine Puffer abgespielt wird, der nächste Puffer mit Daten versorgt und die nächsten Werte für *AUDxLC* und *AUDxLEN* entsprechend gesetzt. Dies erlaubt die Wiedergabe eines beliebig großen Sample-Datenfeldes.

```
; playfile.asm


opt d+

INCDIR      "Include:"
INCLUDE     'exec/exec_lib.i'
INCLUDE     'exec/memory.i'
INCLUDE     'exec/interrupts.i'
INCLUDE     'libraries/dos.i'
INCLUDE     'libraries/dos_lib.i'
INCLUDE     'hardware/custom.i'
INCLUDE     'hardware/dmabits.i'
INCLUDE     'hardware/intbits.i'
INCLUDE     'asmsupp.i'

CUSTOM_ADR  EQU     $DFF000

BUFLEN      EQU     $1000

Begin:
            subq.l      #1,d0                   ; Parameterstring
            clr.b       0(a0,d0.l)              ; am Ende mit 0 versehen
            movea.l     a0,a2

            move.l      4.w,a6                  ; ExecBase
            lea         DLName,a1
            moveq       #0,d0
            CALLSYS     OpenLibrary             ; DOS-Library öffnen
            move.l      d0,DOSBase
            beq         End
```

Taken from Amiga-Manias-Website

```
        movea.l   a2,a0
        bsr       LoadFile                  ; File laden
        move.l    a0,MemPtr                 ; Zeiger auf File im Speicher
        beq       End1
        movea.l   a0,a1
        move.l    d0,MemSize                ; Speicherlänge
        adda.l    d0,a0
        move.l    a0,MemEnd                 ; Ende des Speicherbereiches
        move.l    #BUFLEN,d1
        cmp.l     d1,d0                     ; größer als Puffer?
        ble.s     NoSwtch                   ; nein: ActPtr = MemPtr
        adda.w    d1,a1                     ; sonst eine Pufferlänge weiter

NoSwtch
        move.l    a1,ActPtr

      ; durch getrennte Puffer können kleinere Speicherbereiche
      ; genutzt werden

        move.l    #BUFLEN,d0
        move.l    #MEMF_PUBLIC!MEMF_CHIP,d1
        CALLSYS   AllocMem                  ; Speicher für Puffer1 reservieren
        move.l    d0,Buf1
        beq       End2

        move.l    #BUFLEN,d0
        move.l    #MEMF_PUBLIC!MEMF_CHIP,d1
        CALLSYS   AllocMem                  ; Speicher für Puffer2 reservieren
        move.l    d0,Buf2
        beq       End3

        move.l    MemPtr(PC),a0             ; Puffer 1 vorbereiten
        move.l    Buf1(PC),a1
        move.w    #(BUFLEN/4)-1,d0

CopyB1
        move.l    (a0)+,(a1)+
        dbra      d0,CopyB1

        lea       MyInt(PC),a1
        moveq     #INTB_AUD0,d0
        CALLSYS   SetIntVector              ; Audio-Interrupt setzen
        move.l    d0,OldInt
        beq.s     End4

        lea       CUSTOM_ADR,a2
        move.l    Buf1(PC),ac_ptr+aud0(a2); Puffer 1 zuerst benutzt
        move.l    #BUFLEN,d0
        cmp.l     MemSize(PC),d0
        blt.s     NormLen
        move.l    MemSize(PC),d0
```

Taken from Amiga-Manuals-Website

```
NormLen
        asr.w     #1,d0
        move.w    d0,ac_len+aud0(a2)          ; Pufferlänge in Wörtern
        move.w    #180,ac_per+aud0(a2)        ; Wiedergaberate  X Hz
        move.w    #64,ac_vol+aud0(a2)         ; volle Lautstärke

        move.w    #INTF_SETCLR!INTF_AUD0,intena(a2)     ; Interrupt erlauben
        move.w    #DMAF_SETCLR!DMAF_AUD0,dmacon(a2)     ; DMA einschalten

WaitL1
        btst      #6,$bfe001                  ; auf linke Maustaste warten
        bne.s     WaitL1


        move.w    #DMAF_AUD0,dmacon(a2)       ; DMA ausschalten
        move.w    #INTF_AUD0,intena(a2)       ; Interrupt verbieten

        move.l    OldInt(PC),a1
        moveq     #INTB_AUD0,d0
        CALLSYS   SetIntVector                ; alten Interrupt wiederherstellen

End4
        move.l    Buf2(PC),a1                 ; Puffer 1 und 2 zurückgeben
        move.l    #BUFLEN,d0
        CALLSYS   FreeMem

End3
        move.l    Buf1(PC),a1
        move.l    #BUFLEN,d0
        CALLSYS   FreeMem

End2
        move.l    MemPtr(PC),a1
        move.l    MemSize(PC),d0
        CALLSYS   FreeMem

End1
        move.l    DOSBase(PC),a1
        CALLSYS   CloseLibrary

End
        rts


      ; Die folgenden Register dürfen in einem Interrupt-Handler frei
      ; verwendet (überschrieben) werden, enthalten bei Einsprung in
      ; den Handler jedoch einige nützliche Werte:
      ;
      ;   d0    enthält Müll
      ;   d1    Und-Verknüpfung von INTENA u. INTREQ-Register
      ;   a0    Zeiger auf Customchip-Register ($DFF000)
```

Taken from Amiga-Manuals-Website

```
        ;    a1    Inhalt des is_data-Feldes der Interrupt-Struktur
        ;    a5    Inhalt des is_code-Feldes der Interrupt-Struktur
        ;    a6    ExecBase


IntRoutine:

        move.l    a2,-(a7)                    ; noch ein Adreßregister benötigt

        move.w    #0,color(a0)

        move.w    #INTF_AUD0,intreq(a0)       ; Int.-Anforderung löschen !!!

        move.w    (a1),d0                     ; ActBuf
        eor.w       #4,d0
        move.w    d0,(a1)                     ; auf anderen Puffer umschalten

        lea       Buf1-MyData(a1),a5
        movea.l   0(a5,d0.w),a5               ; Adresse des neuen Puffers
        move.l    a5,aud0+ac_ptr(a0)          ; als Datenzeiger eintragen

        movea.l   ActPtr-MyData(a1),a2        ; Zeiger auf die nächsten Daten
        move.l    MemEnd-MyData(a1),d0
        sub.l     a2,d0                       ; Länge der restlichen Daten
        cmp.l     #BUFLEN,d0                  ; größer als Puffer?
        bgt.s     NoRestart
        movea.l   MemPtr-MyData(a1),a6        ; ja:  ActPtr  := MemPtr
        bra.s     SetStart                    ; AUDXLEN := MemEnd - ActPtr

NoRestart
        move.w    #BUFLEN,d0                  ; nein:ActPtr  := ActPtr + BUFLEN
        movea.l   a2,a6                       ; AUDXLEN := BUFLEN
        adda.w    d0,a6

SetStart
        asr.w     #1,d0                       ; Länge durch 2 = Wörter
        move.w    d0,aud0+ac_len(a0)          ; in AUDXLEN
        move.l    a6,ActPtr-MyData(a1)        ; ActPtr aktualisieren

        asr.w     #1,d0                       ; nochmal durch 2 = Langwörter
        subq.w    #1,d0                       ; minus 1 (dbra!)

CopyBn
        move.l    (a2)+,(a5)+                 ; nächste Daten in Puffer kopieren
        dbra      d0,CopyBn

        move.l    (a7)+,a2                    ; a2 zurückholen
        rts
```

Taken from Amiga-Manuals-Website

```
        ; Interrupt-Strukturen müssen stets auf einer Langwort-Grenze
        ; beginnen. Fügen sie dazu ggf. anstatt "cnop 0,4" die
        ; entsprechende Direktive ihres Assemblers ein.

          cnop      0,4

MyInt:
        ds.l    1
        ds.l    1
        dc.b    NT_INTERRUPT
        dc.b    0
        dc.l    IName

        dc.l    MyData
        dc.l    IntRoutine

OldInt
        ds.l    1

IName
        dc.b    "Audio-Interrupt",0

DLName  DOSNAME                                 ; "dos.library"


MyData:
        ActBuf  ds.w        1

        ActPtr  ds.l        1

        MemPtr  ds.l        1

        MemEnd  ds.l        1

        MemSize ds.l        1

        Buf1    ds.l        1

        Buf2    ds.l        1


LoadFile:
        movem.l     d1-d7/a1-a6,-(a7)
        movea.l     a0,a3
        move.l      a0,d1                       ; Zeiger auf Filenamen
        moveq       #ACCESS_READ,d2             ; Zugriffsart lesen
        LINKSYS     Lock,DOSBase
        move.l      d0,d4                       ; File vorhanden?
        beq         LFErr1
```

Taken from Amiga-Manuals-Website

```
        move.l    d4,d1
        lea       MyFileInfoBlock(PC),a2
        move.l    a2,d2
        LINKSYS   Examine,DOSBase                ; FileInfoBlock-Strukur holen
        move.l    d4,d1
        move.l    d0,d4
        LINKSYS   UnLock,DOSBase
        tst.l     d4
        beq       LFErr1

        tst.l     fib_EntryType(a2)              ; File oder Directory?
        bpl       LFErr1

        move.l    fib_Size(a2),d4                ; Länge des Files
        move.l    d4,d0
        moveq     #MEMF_PUBLIC,d1
        move.l    a6,-(a7)
        move.l    4.w,a6
        CALLSYS   AllocMem                       ; Speicher reservieren
        move.l    (a7)+,a6
        move.l    d0,d6
        beq.s     LFErr1

        move.l    a3,d1                          ; File öffnen
        move.l    #MODE_OLDFILE,d2
        LINKSYS   Open,DOSBase
        move.l    d0,d5
        beq.s     LFErr2

        move.l    d5,d1                          ; Daten einlesen
        move.l    d6,d2
        move.l    d4,d3
        LINKSYS   Read,DOSBase
        cmp.l     d0,d3
        bne.s     LFErr3

        move.l    d5,d1                          ; File schließen
        LINKSYS   Close,DOSBase

        movea.l   d6,a0                          ; Startadresse
        move.l    d4,d0                          ; Länge des Files

LFEnd
        movem.l   (a7)+,d1-d7/a1-a6
        rts


LFErr3
        move.l    d5,d1
        LINKSYS   Close,DOSBase
```

Taken from Amiga-Manuals-Website

```
LFErr2
        movea.l   d6,a1
        move.l    d4,d0
        move.l    a6,-(a7)
        CALLSYS   FreeMem
        move.l    (a7)+,a6

LFErr1
        suba.l    a0,a0
        moveq     #0,d0
        bra.s     LFEnd

        cnop      0,4

MyFileInfoBlock
        ds.b      fib_SIZEOF

        DOSBase    ds.l       1
```

2.2.4 Modulation

Um eine noch größere Bandbreite an Klangeffekten zur Verfügung zu stellen, ist in der Audiohardware des Amiga auch eine Möglichkeit vorgesehen, die Tonausgabe eines Audiokanals durch einen anderen Kanal zu modulieren. Die Modulation kann sich dabei auf zwei Parameter beziehen: Zum ersten kann die Lautstärke verändert werden, was durch eine kontinuierliche Änderung des Wertes im *AUDxVOL*-Register erzielt wird. Dadurch kann das automatische An- oder Abschwellen (Hüllkurve!) oder auch eine zyklische Lautstärkenänderung (Tremolo-Effekt) erreicht werden. Zweitens kann auch die Frequenz des wiedergegebenen Tones moduliert werden, indem der Inhalt des *AUDxPER*-Registers verändert wird. Wird dies zyklisch durchgeführt, so ergibt sich der bekannte Vibrato-Effekt.

Soll die Wiedergabe eines Kanals moduliert werden, so muß man dafür einen zweiten Audiokanal verwenden, der dann nicht mehr zur Tonwiedergabe verwendet werden kann. Dies kommmt daher, daß die von der DMA gelieferten Daten eines Kanals im Modulations-Modus (Attack Mode) nicht mehr an den D/A-Wandler dieses Kanals geschickt werden, sondern in das *AUDxVOL*-oder *AUDxPER*-Register desjenigen Kanals, der moduliert wird. Die Daten, die von dem modulierenden Kanal verarbeitet werden, bestehen jetzt also nicht mehr aus Sample-, sondern aus Lautstärke- oder Periodenwerten, die bei jedem Ausgabe-Zyklus des modulierenden Kanals in das entsprechende Register geschrieben werden. Hierbei werden die DMA-Datenwörter nicht mehr als eine Kombination von zwei Bytes betrachtet, sondern als nur ein »Modulator«-Wort, das jeweils einen 7-Bit-Wert für das *AUDxVOL*- oder einen 15-Bit-Wert zur Übertragung in das *AUDxPER*-Register enthalten kann. Insgesamt lassen sich drei verschiedene Modulations-Betriebsarten unterscheiden:

❑ Lautstärke-Modulation – In dieser Betriebsart werden die von der DMA gelieferten Datenwörter bei jedem Herunterzählen des Periodenzählers in das *AUDxVOL*-Register des zu modulierenden Kanals geschrieben. Ein Modulator-Wort hat hier folgendes Format:

Taken from AmigaManuals-Website

Bit	Funktion
15–7	unbenutzt
6–0	Lautstärkeregister-Wert V6 – V0

Die einzelnen Modulatorwörter müssen im Speicher in der normalen Reihenfolge ihrer Ausgabe angeordnet sein:

Wort-Nr.	Funktion
Wort 1	1. Lautstärkewert
Wort 2	2. Lautstärkewert
Wort 3	3. Lautstärkewert
...	

❑ Frequenz-Modulation – In dieser Betriebsart werden die von der DMA gelieferten Datenwörter bei jedem Herunterzählen des Periodenzählers in das *AUDxPER*-Register des zu modulierenden Kanals geschrieben. Ein Modulator-Wort hat hier folgendes Format:

Bit	Funktion
15–0	Periodenregister-Wert P15 – P0

Die einzelnen Modulatorwörter müssen auch hier in der normalen Reihenfolge ihrer Ausgabe angeordnet sein:

Wort-Nr	Funktion
Wort 1	1. Periodenwert
Wort 2	2. Periodenwert
Wort 3	3. Periodenwert
...	

❑ Kombinierte Lautstärke- und Frequenz-Modulation – In dieser Betriebsart werden die von der DMA gelieferten Datenwörter abwechselnd in *AUDxVOL*- und das *AUDxPER*-Register des zu modulierenden Kanals geschrieben. Eine dafür geeignete Datentabelle muß hierbei folgendes Format haben:

Wort-Nr.	Funktion
Wort 1	1. Lautstärkewert
Wort 2	1. Periodenwert
Wort 3	2. Lautstärkewert
Wort 4	2. Periodenwert
Wort 5	3. Lautstärkewert
...	

Taken from Amiga-Manuals-Website

Dasjenige Register, in dem die Modulationsmodi der einzelnen Kanäle angegeben werden können, ist das *ADKCON*-Register. In diesem Register sind jedem Kanal 2 Bit zugeordnet: Eines, das angibt, ob der betreffende Kanal den mit der jeweils nächsthöheren Nummer in seiner Lautstärke modulieren soll, und entsprechend eines für die Frequenzmodulation. Sind beide Bits gesetzt, so werden Lautstärke- und Frequenz-Modulation kombiniert.

Hieraus ist zu sehen, daß der Kanal, der von einem anderen moduliert werden soll, leider nicht frei gewählt weren kann. Es kann vielmehr immer nur derjenige Kanal von einem anderen moduliert werden, der die zu diesem nächsthörere Ordnungsnummer hat, also kann z.B Kanal 0 nur Kanal 1 modulieren, Kanal 1 nur Kanal 2 und so fort. Dies hat auch zur Folge, daß Kanal 3 nicht zur Modulation eingesetzt werden kann, da es keinen Kanal mit höherer Ordnungsnummer gibt. Werden die jeweiligen Bits für Kanal 3 im *ADKCON*-Register gesetzt, so schaltet dies lediglich die Audio-Ausgabe dieses Kanals ab.

Das *ADKCON*-Register hat die folgende Bitbelegung:

Bit	Name	Funktion
15	SETCLR	1 = alle 1-Bits werden auf 1 gesetzt, alle anderen bleiben unberührt 0 = alle 1-Bits werden auf 0 gesetzt, alle anderen bleiben unberührt
14–8		Vom Disk-Controller verwendet
7	USE3PN	Audio-Kanal 3 moduliert nichts
6	USE2P3	Audio-Kanal 2 moduliert Periode Kanal 3
5	USE1P2	Audio-Kanal 1 moduliert Periode Kanal 2
4	USE0P1	Audio-Kanal 0 moduliert Periode Kanal 1
3	USE3PN	Audio-Kanal 3 moduliert nichts
2	USE2V3	Audio-Kanal 2 moduliert Volume Kanal 3
1	USE1V2	Audio-Kanal 1 moduliert Volume Kanal 2
0	USE0V1	Audio-Kanal 0 moduliert Volume Kanal 1

Das folgende Beispielprogramm verwendet die hier vorgestellten Möglichkeiten zur Erzeugung eines sogenannten Leslie-Effekts. Eine Leslie-Box, bekannt aus den kruden Orgel-Orgien der Rockmusik der 60er und 70er Jahre, besteht aus zwei auf einer Achse montierten Hörnern, die man in verschiedenen Geschwindigkeiten rotieren lassen kann. Dies ergibt durch die unterschiedlichen Relativgeschwindigkeiten zum Ohr außer einer Lautstärke- auch eine Frequenzmodulation (Doppler-Effekt). Dies wird in dem Programm dadurch nachgeahmt, daß Lautstärke und Tonhöhe zyklisch und jeweils gegenläufig verschoben werden. Leider ist die Stärke des Doppler-Effekts (die Modulationstiefe) ganz wesentlich von der Umdrehungsgeschwindigkeit der Hörner abhängig, was bedeutet, daß man eigentlich für jede eine eigene Tabelle bräuchte oder zumindest bei jeder Änderung eine neue berechnen müßte (die Integration einer Floating-Point-Behandlung hatte wohl den Rahmen eines Demoprogramms gesprengt). So klingt der Effekt auch nur in einem kleinen Bereich von Modulationsgeschwindigkeiten einigermaßen realistisch. Wenn man allerdings ein wenig mit den Modulationstabellen experimentiert, wird man sicher noch einige erstaunliche Variationen dieses Effekts erzielen können, bis hin zum dreidimensionalen Raumklang-Effekt.

Amiga-Manuals-Website

Das Programm benutzt Kanal 0 zur Modulation von Kanal 1 und Kanal 2 für Kanal 3. Beide modulierenden Kanäle, 0 und 2, verwenden nur eine Tabelle, wobei die Ausgabe jedoch um eine halbe Tabellenlänge versetzt erfolgt. Dies wird dadurch erreicht, daß zuerst die Adresse der Tabellenmitte in das Zeiger- die halbe Tabellenlänge in das Zählerregister von Kanal 2 geschrieben werden, und dann die DMA gestartet wird. Dann werden die eigentlichen Werte der Tabelle angegeben, die erst dann in Zeiger und Zähler übernommen werden, wenn sich die DMA von Kanal 0 bereits bei der Mitte der Tabelle befindet, wodurch beide von nun an entsprechend versetzt stattfinden.

Nach der Übergabe aller Parameter und dem Start der DMA werden ständig beide Maus-Knöpfe abgefragt. Beim Druck auf den rechten Knopf wird die Modulationsgeschwindigkeit erhöht, beim linken verringert. Drückt man beide Knöpfe, so endet das Programm.

```
;leslie.asm


INCLUDE     'exec/exec_lib.i'
INCLUDE     'exec/memory.i'
INCLUDE     'hardware/custom.i'
INCLUDE     'hardware/adkbits.i'
INCLUDE     'hardware/dmabits.i'
INCLUDE     'hardware/intbits.i'
INCLUDE     'asmsupp.i'


CUSTOM_ADR    EQU    $DFF000

MAXMODPER     EQU    65535
MINMODPER     EQU    1400

          move.l     4.w,a6
          move.l     #dlen,d0
          moveq      #MEMF_PUBLIC!MEMF_CHIP,d1
          CALLSYS    AllocMem
          move.l     d0,buffer
          beq        End
          lea        sampledata(PC),a0
          movea.l    d0,a1
          moveq      #(dlen/2)-1,d1

CopyBuf
          move.w     (a0)+,(a1)+
          dbra       d1,CopyBuf
          lea        CUSTOM_ADR,a2
          move.l     buffer(PC),d0
          move.l     d0,ac_ptr+aud1(a2)              ; Abspielbereich Kanal 1
          move.l     d0,ac_ptr+aud3(a2)              ; Abspielbereich Kanal 3
          move.w     #(sdlen/2),ac_len+aud1(a2)      ; Datenlänge in Wörtern
          move.w     #(sdlen/2),ac_len+aud3(a2)
          move.w     #223,ac_per+aud1(a2)            ; Startfrequenz 1000 Hz
          move.w     #223,ac_per+aud3(a2)
```

Taken from Amiga-Manuals-Website

```
        move.w    #64,ac_vol+aud1(a2)              ; Startlautstärke
        move.w    #64,ac_vol+aud3(a2)

        addi.l    #sdlen,d0                        ; = moddata
        move.l    d0,d1
        move.l    d0,ac_ptr+aud0(a2)               ; Modulationsdaten Kanal 0
        addi.l    #mdlen/2,d1                      ; Mod.-Phase Kanal 2 =
        move.l    d1,ac_ptr+aud2(a2)               ; Mod.-Phase Kanal 0 + pi
        move.w    #mdlen/2,ac_len+aud0(a2)
        move.w    #mdlen/4,ac_len+aud2(a2)
        move.l    #$4000,d2
        move.w    d2,ac_per+aud0(a2)
        move.w    d2,ac_per+aud2(a2)

        move.w
#ADKF_SETCLR!ADKF_USE0V1!ADKF_USE2V3!ADKF_USE0P1!ADKF_USE2P3,adkcon(a2)

        move.w    #DMAF_SETCLR!DMAF_AUDIO,dmacon(a2) ; alle 4 Kanäle einsch.

        move.l    d0,ac_ptr+aud2(a2)               ; Bereich Knl 2 korrigieren
        move.w    #mdlen/2,ac_len+aud2(a2)


        move.l    #MAXMODPER,d3
        move.l    #MINMODPER,d4

WaitLM
        btst      #6,$bfe001                       ; linke Maustaste testen
        beq.s     WaitRM
        btst      #2,potgor(a2)                    ; rechte Maustaste testen
        bne.s     WaitLM

        subq.l    #1,d2
        cmp.l     d4,d2
        bpl.s     NoCorr
        move.l    d4,d2

NoCorr
        bra.s     SetMPer

WaitRM
        btst      #2,potgor(a2)                    ; beide Tasten = Ende
        beq.s     End1
        addq.l    #1,d2
        cmp.l     d2,d3
        bpl.s     SetMPer
        move.l    d3,d2

SetMPer
        move.w    d2,ac_per+aud0(a2)
        move.w    d2,ac_per+aud2(a2)
        bra.s     WaitLM
```

Taken from Amiga-Manuals-Website

```
End1
        move.w      #DMAF_AUDIO,dmacon(a2)          ; DMA ausschalten

        move.w      #ADKF_USE0V1!ADKF_USE2V3!ADKF_USE0P1!ADKF_USE2P3,adkcon(a2)

        move.l      buffer(PC),a1                   ; Puffer zurückgeben
        move.l      #dlen,d0
        CALLSYS     FreeMem

End:
        rts

sampledata:
        dc.b        0,49
        dc.b        90,117
        dc.b        127,117
        dc.b        90,49
        dc.b        0,-49
        dc.b        -90,-117
        dc.b        -127,-117
        dc.b        -90,-59

moddata:
        dc.w        64,223
        dc.w        64,223
        dc.w        63,223
        dc.w        62,223
        dc.w        61,222
        dc.w        59,222
        dc.w        57,221
        dc.w        55,221
        dc.w        53,220
        dc.w        50,220
        dc.w        47,219
        dc.w        44,219
        dc.w        41,219
        dc.w        38,219
        dc.w        35,219
        dc.w        33,219
        dc.w        30,220
        dc.w        27,220
        dc.w        25,221
        dc.w        23,221
        dc.w        21,222
        dc.w        19,222
        dc.w        18,223
        dc.w        17,223
        dc.w        16,223
```

Taken from Amiga-Manuals-Website

```
        dc.w        16,223
        dc.w        16,223
        dc.w        17,223
        dc.w        18,224
        dc.w        19,224
        dc.w        21,225
        dc.w        23,225
        dc.w        25,226
        dc.w        27,226
        dc.w        30,227
        dc.w        33,227
        dc.w        35,228
        dc.w        38,228
        dc.w        41,228
        dc.w        44,228
        dc.w        47,227
        dc.w        50,227
        dc.w        53,226
        dc.w        55,226
        dc.w        57,225
        dc.w        59,225
        dc.w        61,224
        dc.w        62,224
        dc.w        63,223
        dc.w        64,223

dend:
        dlen    EQU  dend-sampledata            ; Gesamtlänge

        sdlen   EQU  moddata-sampledata         ; Länge der Sample-Daten

        mdlen   EQU  dend-moddata               ; Länge der Modulations-Daten

buffer
        ds.l        1
```

2.2.5 Prozessor-Betrieb

Neben der Wiedergabe durch die DMA gibt es auch noch die Möglichkeit, die benötigten Daten vom Prozessor in das Datenregister der Audioeinheit schreiben zu lassen. Dies wird vor allem dann benötigt, wenn Audio-Datenverabeitung in Echtzeit verlangt wird. Stellen Sie sich beispielsweise vor, Sie haben einen Digitalisierer am Amiga angeschlossen, dessen Werte Sie sofort und möglicherweise verfremdet wieder ausgeben wollen, um den Amiga zum »Effektgerät« umzufunktionieren. Dann müssen vom Prozessor bei jedem Sampling-Zyklus die Daten aus dem Digitizer geholt, ggf. verarbeitet und dann in das Wandlerregister geschrieben werden. Gesteuert wird das durch den Audio-Interrupt, der nun nicht mehr nach der Übernahme der *AUDxPTR*- und *AUDxLEN*-Registerwerte ausgelöst wird, sondern nach jeder Ausgabe des höherwertigen Bytes im Wandlerregister. In diesem Interrupt muß dieses Register wieder mit einem neuen 2-Sample-Wort versorgt und die entsprechende Interruptanforderung gelöscht werden (*AUDx*-Bits im

Taken from Amiga-Manuals-Website

INTREQ-Register auf 0 setzen). Geschieht eines von beiden nicht rechtzeitig, so geht die Audio-Kontrolle des Kanals in den Idle State über.

Das Wandlerregister trägt den Namen *AUDxDAT* und enthält wie erwähnt in den Bits 15–8 das höherwertige, zuerst ausgegebene und in den Bits 7–0 das niederwertige Sample, das während des nächsten Durchlaufs des Periodenzählers ausgegeben wird. Die Adressen lauten:

$DFF0AA [AUD0DAT] Datenregister Kanal 0
$DFF0BA [AUD1DAT] Datenregister Kanal 1
$DFF0CA [AUD2DAT] Datenregister Kanal 2
$DFF0DA [AUD3DAT] Datenregister Kanal 3

Um nun einen solchen Prozesorbetrieb zu realisieren, müssen zunächst das *AUDxPER*- und das *AUDxVOL*-Register entsprechend den gewünschten Werten gesetzt werden. Dann muß nur noch irgendein Wert in das *AUDxDAT*-Register geschrieben werden, um den Vorgang zu starten. Dadurch wird sofort ein Interrupt ausgelöst, der, um korrekt zu arbeiten, natürlich vorher entsprechend initialisiert sein muß, z.B. mit Hilfe der Exec-Routine *SetIntVector()*. Dies wiederholt sich von nun an bei jedem zweiten Durchlauf des Periodenzählers, solange die oben genannten Aktionen im Interrupt rechtzeitig durchgeführt werden und der Interrupt nicht verboten wird (*AUDx*-Bit im *INTRENA*-Register auf 0 setzen).

Da hier bei jedem zweiten Sampling-Zyklus ein Interrupt ausgelöst werden muß, ist diese Methode sehr rechenzeitintensiv, was sich bei höheren Ausgaberaten sehr bald durch eine deutliche Verlangsamung des Amiga bemerkbar macht. Deshalb sollte ihre Verwendung auf Echtzeitanwendungen beschränkt bleiben. Zur Reduzierung des Rechenzeitbedarfs läßt sich jedoch folgender Tip geben: Möchte man ein Programm schreiben, das das Multitasking nicht unbedingt benötigt, so sollte man sich überlegen, ob man den Interrupt wirklich mit SetIntVector initialisieren und ihn damit über das System laufen lassen möchte. Die hierfür zuständigen Systemroutinen weisen nämlich selbst schon einen nicht unerheblichen Rechenzeitbedarf auf, so daß man durch die Umgehung derselben (d.h. wenn man die Interruptvektoren $64 – $78 direkt auf die eigenen Routinen setzt) unter Umständen eine wesentlich höhere Ausbeute an Sampling-Zyklen pro Zeit erzielen kann.

Das Beispielprogramm hierzu nutzt den Prozessor-Betrieb zur direkten Wiedergabe der Daten eines am Drucker-Port angeschlossen A/D-Wandlers. Hierzu wird zunächst der Audio-Interrupt von Kanal 0 initialisiert und dann *AUDxVOL* und *AUDxPER* gesetzt. Durch Schreiben einer 0 in das *AUDxDAT*-Register wird der Interrupt zum erstenmal ausgelöst. In diesem wird der neue Wert aus dem Sampler geholt, was bei den meisten Samplern bereits den nächsten Wandlerzyklus auslöst. Dann wird ein Durchschnittswert für das zuerst ausgegebene Sample gebildet (verringert das Quantisierungsrauschen) und der 2-Byte-Wert in das *AUDxDAT*-Register übergeben. Ein Klick auf die rechte Maustaste beendet das Programm.

```
; direct.asm


INCLUDE    'exec/exec_lib.i'
INCLUDE    'exec/interrupts.i'
```

```
INCLUDE     'hardware/custom.i'
INCLUDE     'hardware/cia.i'
INCLUDE     'hardware/intbits.i'
INCLUDE     'asmsupp.i'


CIAA_ADR    EQU         $bfe001
CIAB_ADR    EQU         $bfd000


            lea         $dff000,a2

            lea         CIAA_ADR,a3             ; Druckerport auf Eingang
            sf          ciaddrb(a3)

            move.w      #64,aud0+ac_vol(a2)     ; volle Lautstärke
            move.w      #223,aud0+ac_per(a2)    ; Sampling-Rate 16 kHz

            lea         MyInt(PC),a1            ; Interrupt initialisieren
            moveq       #INTB_AUD0,d0
            move.l      a6,-(a7)
            movea.l     4.w,a6
            jsr         _LVOSetIntVector(a6)
            movea.l     (a7)+,a6
            tst.l       d0
            beq         End
            move.l      d0,OldInt

            move.w      #INTF_SETCLR!INTF_AUD0,intena(a2)    ; Interrupt erlauben

            move.w      #0,aud0+ac_dat(a2)      ; Interrupt auslösen

WaitL1
            btst        #6,(a3)                 ; auf Maus warten
            bne.s       WaitL1
            move.w      #INTF_AUD0,intena(a2)   ; Interrupt verbieten
            move.l      OldInt(PC),a1           ; alten Interrupt wiederherstellen
            moveq       #INTB_AUD0,d0
            move.l      a6,-(a7)
            movea.l     4.w,a6
            jsr         _LVOSetIntVector(a6)
            movea.l     (a7)+,a6

End
            rts


            ; Die folgenden Register dürfen in einem Interrupt-Handler frei
            ; verwendet (überschrieben) werden, enthalten bei Einsprung in
            ; den Handler jedoch einige nützliche Werte:
            ;
            ; d0 enthält Müll
```

Taken from Amiga-Manuals-Website

```
        ;       d1      Und-Verknüpfung von INTENA u. INTREQ-Register
        ;       a0      Zeiger auf Customchip-Register ($DFF000)
        ;       a1      Inhalt des is_data-Feldes der Interrupt-Struktur
        ;       a5      Inhalt des is_code-Feldes der Interrupt-Struktur
        ;       a6      ExecBase


IntRoutine:
        move.w          #INTF_AUD0,intreq(a0)       ; Interrupt löschen
        clr.l           d0
        move.b          (a1),d0                     ; neues Sample holen
        sub.w           #$80,d0                     ; auf Zweierkompl.-Form bringen
        lea             lsample(PC),a5
        move.w          (a5),d1                     ; vorheriges Sample
        move.w          d0,(a5)                     ; neues zwischenspeichern
        add.w           d0,d1                       ; Mittelwert bilden
        lsl.w           #7,d1                       ; : 2; * 256 (->higher Byte)
        move.b          d0,d1                       ; kombinieren
        move.w          d1,aud0+ac_dat(a0)
        rts


        cnop            0,4

MyInt:
        ds.l            1
        ds.l            1
        dc.b             NT_INTERRUPT
        dc.b            0
        dc.l            IName
        dc.l            CIAA_ADR+ciaprb             ; Drucker-Port
        dc.l            IntRoutine

OldInt
        ds.l            1

IName
        dc.b            "Audio-Interrupt",0

lsample
        ds.w            1
```

2.2.6 Audio-Zustandsdiagramm

(Aus: Heiko Knappe, Amiga-Sounder, Markt&Technik)

Die Audio-Erzeugung des Amiga kann als endlicher Automat beschrieben werden. Ein solcher Automat besitzt endlich viele verschiedene diskrete Zustände, die ineinander übergehen können. Diese Übergänge erfolgen, wenn in einem Zustand ein bestimmtes Ereignis (zum Beispiel ein Eingabezeichen) eintritt. Dabei können bei einem Zustandswechsel vom endlichen Automaten zum Beispiel Zeichen ausgegeben werden. Die Automatentheorie ist ein sehr interessantes Gebiet, das für die Informatik fundamentale Bedeutung besitzt. Denn der endliche Automat ist nur der

Taken from Amiga-Mania's Website

einfachste Typ: Es gibt Kellerautomaten, linear beschränkte Automaten oder Turingmaschinen. Alle diese Maschinen sind durch einen eigenen Sprachtyp und eine eigene Grammatik gekennzeichnet. Mit Ihnen kann fast alles beschrieben werden. – Der Computer nach dem von Neumannschen Modell, also auch der Amiga, bereitet keine Schwierigkeiten.

Allerdings reicht dafür nicht mehr ein endlicher Automat.

Dieser Abschnitt beschließt die Erklärungen zur Erzeugung von Tönen mit dem Amiga. Er dient in erster Linie der Zusammenfassung bisher vorgestellter Möglichkeiten und soll das Gesamtkonzept der Audio-Hardware demonstrieren.

Die Amiga-Audio-Maschine

Für jeden der vier Audiokanäle gibt es eine solche Maschine. Sie wird durch einen endlichen Automaten (siehe oben) mit acht Zuständen (000, 001, 010, 011, 100, 101, 110 und 111) beschrieben und wird mit dem Systemtakt von 3,58 MHz getaktet. Die Zustände 100, 110 und 111 haben keine besondere Bedeutung und führen mit spontanen Übergängen in den Grundzustand 000.

Aus dem Grundzustand (000) können zwei Übergänge erfolgen. Der eine ist für die Interrupt gesteuerte Audioausgabe gedacht, bei der die Audiodaten vom Prozessor zur Verfügung gestellt werden. Der andere Übergang erfolgt bei einer DMA-Ausgabe, bei der der Spezial-Chip »Agnus« die Daten heranschafft. Die eigentliche Ausgabe erfolgt dann in der sogenannten Hauptschleife, die aus den Zuständen 010 und 011 besteht.

Interrupt-gesteuerte Audioausgabe

Hier erfolgt der Übergang in die Hauptschleife (010 und 011), sobald Daten vom Prozessor in das AUDxDAT-Register geschrieben wurden. Im Zustand 010 wird das obere Byte (penhi) des Datenworts ausgegeben und bei 011 das untere (penhi). Ein Übergang von 010 nach 011 und umgekehrt erfolgt immer, wenn der Periodenzähler auf 1 heruntergezählt wurde (perfin). Dann wird die Periode neu geladen (percntrld). Wurde ein Datenwort ausgegeben, d.h., befindet sich der Automat im Zustand 011, so wird nach Herunterzählen des Periodenzählers (perfin) ein Audio-Interrupt ausgelöst (AUDxIR). Dieser muß umgehend (rechtzeitig!) vom Prozessor, d.h., einer Interruptroutine beantwortet werden (AUDxIP), damit nicht in den Grundzustand zurückgekehrt wird. Dieses rechtzeitige Rücksetzen macht es notwendig, in der eigenen Interrupt-Routine mit einem der ersten Befehle den Interrupt zu beantworten, da sonst nach mehreren Durchläufen die Hauptschleife verlassen wird. Dieses ist auf eine immer größere Zeitdifferenz zwischen Anforderung und Rückmeldung zurückzuführen. Die Interrupt-gesteuerte Audioausgabe ist weiter oben bereits ausführlich mit Programmbeispielen vorgestellt worden.

DMA-gesteuerte Audioausgabe

Sobald der Audio-DMA gestartet wird (AUDxEN-Bits in DMACON werden gesetzt / AUDxON), geht der Automat vom Grundzustand (000) in 001 über. Dabei erhält »Agnus« die Aufforderung, mit dem Datentransport zu beginnen (AUDxDR). Wegen der internen Architektur von »Agnus« entspricht das erste gesendete Datenwort nicht dem ersten auszugebenden Zeichen und darf deshalb nicht ausgegeben werden. Aus diesem Grund wurden die Zwischenzustände 001 und 101 eingeführt, bevor die Hauptschleife erreicht wird. Sobald nun dieses erste Wort

angekommen ist, wird der Zustand 010 erreicht. Beim nächsten, d.h., dem ersten Wert der Wellenform, erreicht die »Maschine« die Hauptschleife, die erst beim Zurücksetzen des Audio-DMA verlassen wird. Der Wort-Zähler erniedrigt sich bei jedem ankommenden Wort um eins (lencount), bis er auf 1 heruntergezählt ist (lenfin). Dann ergeht mit der normalen Datenanforderung der Befehl an »Agnus«, den zwischengespeicherten Längen-Zähler (lencntrld) neu zu laden und den aktuellen Datenzeiger auf das erste Wort der Wellenform zurückzusetzen. Außerdem wird, sobald die Ausgabe des letzten Datenworts der Wellenform begonnen hat, ein Interrupt ausgelöst. DMA- und Neustart-Anforderungen werden einmal pro horizontaler Video-Zeile zum Spezial-Chip »Agnus« übertragen. Die Daten sind dann etwa 14 Taktzyklen später verfügbar.

Modulation

Bei den Modulationsmodi (im Register ADKCON bestimmt) wird etwas anders vorgegangen: Bei Amplitudenmodulation erfolgen die Datenanforderungen wie üblich bei dem Übergang von Zustand 011 nach 010. Dagegen werden die Anforderungen bei der Frequenzmodulation beim Zustandswechsel von 010 nach 011 ausgelöst. Werden sowohl die Amplitude als auch die Frequenz moduliert, so werden bei beiden Übergängen Daten angefordert.

Notwendige Signaldefinitionen für das Zustandsübergangsdiagramm

Die groß geschriebenen Bezeichnungen stehen für externe Signale, die klein geschriebenen für interne. Wie bisher üblich, steht das »x« zum Beispiel in AUDxON für die Nummer des gewünschten Kanals, also 0, 1, 2 oder 3.

AUDxON	DMA-Bits für den Audiokanal x aus dem DMA-Kontrollregister DMACON.
AUDxIP	Audio-Interrupt-Anforderung von Audiokanal x. Dieser erwartet die Anwort des Systems und ein neues Datenwort für die Ausgabe. Dieser Vorgang ist für die Direkt-Ausgabe von Bedeutung!
AUDxIR	Audio-Interrupt-Meldung des Audiokanals x, falls die Ausgabe eines Datenwortes beendet ist. Diese Meldung hängt unmittelbar mit AUDxIP zusammen und ist vor allem für die Direktausgabe von Bedeutung!
AUDxDAT	Signalisiert die Übergabe eines neuen Datenworts aus dem 16 Bit-AUDxDAT-Register in den internen Audio-Kanal.
AUDxDR	DMA hat das nächste Datenwort in das AUDxDAT-Register geschrieben, das 'Agnus' nun übernehmen soll.
AUDxDSR	DMA verlangt von 'Agnus', daß der Zeiger auf das aktuelle Datenwort auf den ersten Wert der Wellendaten im Speicher zurückgesetzt wird. Das heißt die Wellenform wurde einmal ausgegeben und soll nun wieder von vorne beginnen.
AUDxAV	Amplitudenmodulation – Datenwort nicht zum D/A-Wandler schicken, sondern als Lautstärke für den nächsten Kanal verwenden.
AUDxAP	Frequenzmodulation – Datenwort nicht zum D/A-Wandler schicken, sondern als Periode für den nächsten Kanal verwenden.

intreq2	Vorbereitung zur Interrupt-Anforderung. Anforderung erfolgt nach dem nächsten Übergang von Zustand 011 nach 010 bei normaler Ausführung.
dmasen	dmasen*AUDxDR=AUDxDSR – dmasen besagt, daß für 'Agnus' zum erstenmal ein Datenwort angefordert wird (Übergang von 000 nach 001).
percntrld	Nachladen der in AUDxPER gespeicherten Sampling-Periode (Back-up). Typisch wird dieser Wert vom Prozessor bei der Audio-Initialisierung gesetzt, kann aber auch bei der Frequenzmodulation vom DMA bestimmt werden.
percount	Erniedrigt den Perioden-Zähler um 1.
perfin	Zeigt an, daß der Perioden-Zähler bis auf 1 heruntergezählt ist und damit die Warteschleife beendet ist.
lencntrld	Nachladen der in AUDxLEN gespeicherten Wortanzahl (Back-Up).
lencount	Erniedrigt den Wort-Zähler um 1.
lenfin	Der Wort-Zähler wurde auf 1 heruntergezählt. Das augenblickliche Zeichen ist damit das letzte der Wellenform.
volcntrld	Lädt die Lautstärke aus AUDxVOL (Back-up).
pbufld1	Übernehmen der Daten aus AUDxDAT.
pbufld2	Übernehmen der Daten aus AUDxDAT beim Übergang von Zustand 010 in 011 bei Frequenzmodulation (entspricht sonst 'pbufld1').
penhi	Die oberen 8 Bit (d.h. das obere Byte) des Datenworts liegen am D/A-Wandler an. penhi steht entsprechend für die Ausgabe des unteren Bytes.
napnav	AUDxAV+{|AUDxAV|*|AUDxAP|} – Dieser Ausdruck bedeutet, daß entweder keine Modulation oder Amplutudenmodulation vorliegt. Er ist die Bedingung für normale DMA und Interrupt-Anforderungen.

Beschreibung der Zustandsübergänge

alter Zustand	neuer Zustand	Beschreibung
100	000	[] () spontaner Übergang
110	000	[] () spontaner Übergang
111	000	[] () spontaner Übergang
000	000	[percntrld] () Im Grundzustand wird ständig die Periode aus AUDxPER übernommen.

Taken from Amiga Mania's website

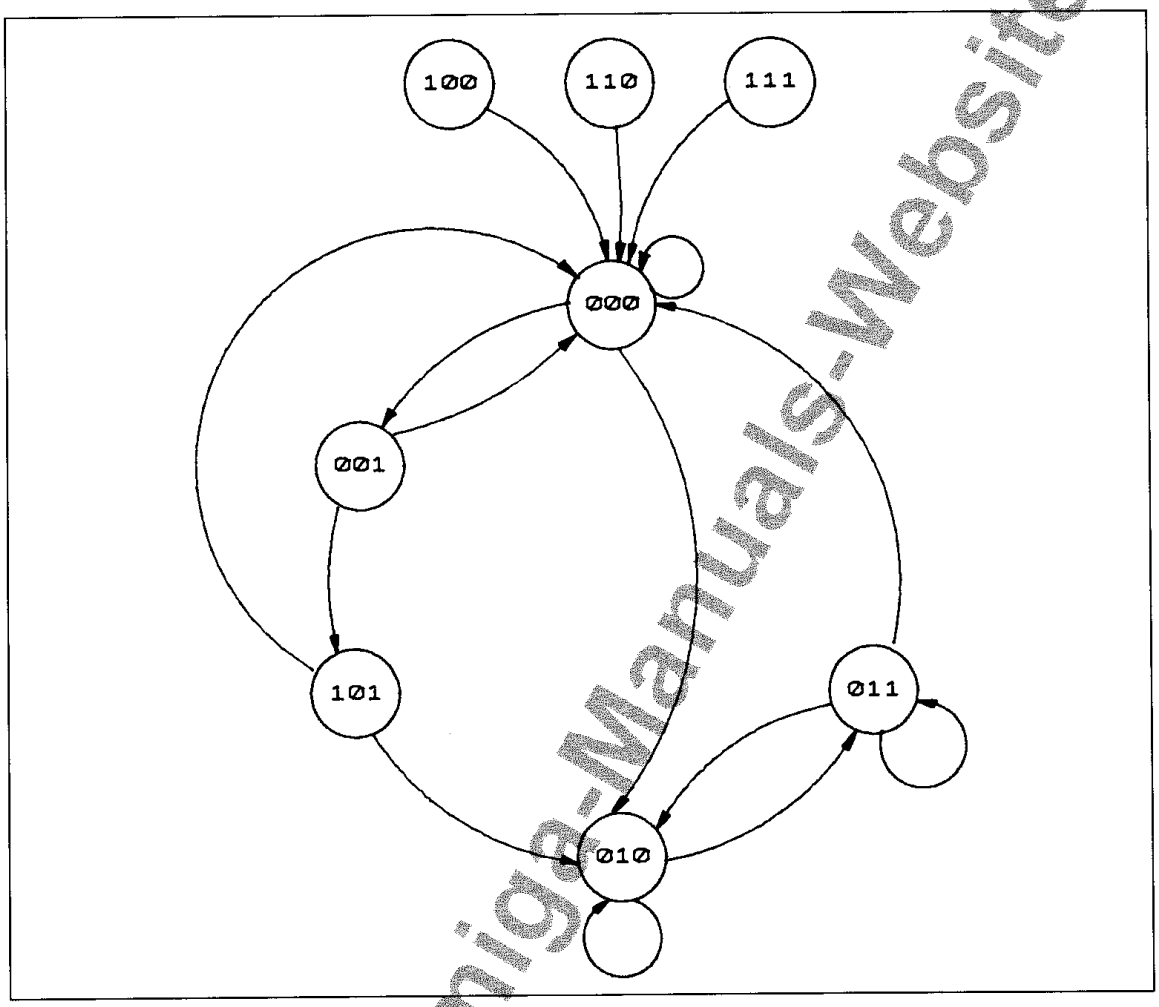


Abb. 2.2: Die Amiga-Audio-Maschine – die Audio-Erzeugung als endlicher Automat

alter Zustand	neuer Zustand	Beschreibung
000	010	[volcntrld,percntrld,pbufldl,AUDxIR] (AUDxDAT*|AUDxON|*|AUDxIP|) Bei diesem Übergang wird die interruptgesteuerte Audioausgabe gestartet. Dazu darf die DMA-Ausgabe nicht angefordert sein, keine Interrupt-Anforderung bestehen und ein Datenwort muß in AUDxDAT geschrieben worden sein. Beim Übergang wird die gewünschte Lautstärke (volcntrld) und Periode (percntrld) übernommen. Der Wert in AUDxDAT wird gelesen und ein Interrupt zur Anforderung des ersten Datenworts ausgelöst.

Taken from Amiga-Manuals-Website

alter Zustand	neuer Zustand	Beschreibung
000	001	[lencntrdl,AUDxDR,dmasen,percntrld] (AUDxON) Die DMA-Ausgabe wird gestartet. Es wird von 'Agnus' das erste Datenwort angefordert (AUDxDR). Außerdem wird die Periode (percntrld) und die Anzahl der Worte (lencntrld) geladen.
001	000	[] (!AUDxON!) Wurde der DMA zwar gestartet, jedoch sofort wieder gestoppt (!AUDxON!), so wird in den Grundzustand zurückgekehrt.
001	101	[AUDxIR,lencount falls !lenfin!] (AUDxON*AUDxDAT) Das erste Datenwort ist eingetroffen (Abfall!) und der DMA ist weiterhin angeschaltet. Beim Übergang wird eine Interruptanforderung ausgelöst (AUDxIR) und die Anzahl der auszugebenden Worte um 1 erniedrigt. (Letzteres nur falls das Ende noch nicht erreicht ist.)
101	000	[] (!AUDxON!) Wurde der DMA zwar gestartet, jedoch nach Eintreffen des ersten Worts wieder gestoppt (!AUDxON!), so wird in den Grundzustand zurückgekehrt.
101	010	[volcntrld,percntrld,pbufldl,AUDxDR falls napnav] (AUDxON*AUDxDAT) Das erste richtige Datenwort ist angekommen. Die DMA-Ausgabe ist noch aktiv. Es wird die Lautstärke (volcntrld), Periode (percntrld) und das neue Datenwort übernommen. Außerdem wird ein neues Wort von 'Agnus' angefordert, falls keine Frequenzmodulation angeschaltet ist.
010	010	[percount,penhi] () Die Ausgabe des oberen Bytes erfolgt solange, bis der Perioden-Zähler auf 1 erniedrigt wurde. Es wird im Systemtakt von 3.58 MHz der Zähler um 1 erniedrigt. Wie lange in diesem Zustand verblieben wird, bestimmt die Ausgabegeschwindigkeit!
010	011	[pubfld2 falls AUDxAP, AUDxDR falls AUDxAP*AUDxON] [percntld,AUDxIR falls intreq2*AUDxAP*AUDxON] [AUDxIR falls AUDxAP*AUDxON] [lencntrld falls lenfin*AUDxON*AUDxDAT] [lencount falls !lenfin!*AUDxON*AUDxDAT] [intreq2 falls lenfin*AUDxON*AUDxDAT] (perfin)

Taken from Amiga Magazin Website

alter Zustand	neuer Zustand	Beschreibung
		Sobald der Perioden-Zähler 1 erreicht hat (perfin), wird mit der Ausgabe des unteren Bytes (Zustand 011) begonnen. Dabei wird der Längen-Zähler um 1 erniedrigt (lencount), falls das Ende der Wellenform noch nicht erreicht ist (lenfin), die Ausgabe mit DMA erfolgt und ein neues Wort im AUDxDAT-Register steht. Sollte jedoch das Ende erreicht sein (lenfin), so wird die zwischengespeicherte Anzahl der Datenworte aus AUDxLEN nachgeladen (lencntrld). Bei DMA-Ausgabe und neuem Wort in AUDxDAT wird intreq2 ausgelöst Bei Frequenzmodulation wird bei diesem Übergang das Datenwort bereits übernommen (pbufld2). Sollte die Ausgabe mit DMA und frequenzmoduliert geschehen, erfolgt eine Anforderung nach einem neuen Datenwort an 'Agnus' (AUDxDR). Falls außerdem intreq2 auftritt, wird außerdem noch ein Audio-Interrupt ausgelöst (AUDxIR). Auf alle Fälle wird die Periode neu aus AUDxPER geladen.
011	011	[percount,penhi] () Die Ausgabe des unteren Bytes erfolgt solange, bis der Perioden-Zähler auf 1 erniedrigt wurde. Es wird im Systemtakt von 3.58 MHz der Zähler um 1 erniedrigt. Wie lange in diesem Zustand verblieben wird, bestimmt die Ausgabegeschwindigkeit!
011	010	[AUDxIR falls intreq2*AUDxON*napnav] [AUDxIR falls napnav*|AUDxON|] [pbufld,percntrld,AUDxDR falls AUDxON*napnav] [lencntrld falls lenfin*AUDxON*AUDxDAT] [lencount falls |lenfin|*AUDxON*AUDxDAT] [intreq2 falls lenfin*AUDxON*AUDxDAT] (perfin*{AUDxON+|AUDxIP|}) Ist der Perioden-Zähler auf 1 heruntergezählt (perfin) und entweder der DMA an (AUDxON) oder wurde die Interrupt-Anforderung (AUDxIR) rechtzeitig beantwortet (AUDxIP), so erfolgt der Übergang in Zustand 010. Der Automat bleibt also in der Haupt-Ausgabe-Schleife. Beim Übergang wird auf alle Fälle das Datenwort (pbufld) und die Periode (percntrld) geladen. Dabei wird der Längen-Zähler um 1 erniedrigt (lencount), falls das Ende der Wellenform noch nicht erreicht ist (|lenfin|), die Ausgabe mit DMA erfolgt und ein neues Wort im AUDxDAT-Register steht. Sollte jedoch das Ende erreicht sein (lenfin), so wird die zwischengespeicherte Anzahl der Datenworte aus AUDxLEN nachgeladen (lencntrld). Bei DMA-Ausgabe und neuem Wort in AUDxDAT wird intreq2 ausgelöst. (Dies entspricht dem Übergang 010 nach 011!)

Taken from Amiga Hardware Reference Manual

alter Zustand	neuer Zustand	Beschreibung
		Die folgenden Aktionen werden ausgeführt, wenn keine Frequenzmodulation ausgeführt wird (napnav): Ein Interrupt wird ausgelöst, falls intreq2 vorliegt und der DMA aktiv ist. Erfolgt die Ausgabe Interrupt-gesteuert (|DMAxON|), so geschieht dies auf jeden Fall. Außerdem wird bei DMA-Betrieb von 'Agnus' das nächste Datenwort angefordert.
011	000	[] (perfin*|{AUDxON+|AUDxIP|}|) Falls der Perioden-Zähler auf 1 erniedrigt worden ist und der DMA ausgeschaltet wurde bzw. keine rechtzeitige Beantwortung der Interrupt-Anforderung erfolgte, wird die Hauptschleife verlassen und in den Grundzustand zurückgekehrt. Mit diesem Übergang ist die Ausgabe beendet.

Die Begriffe in runden Klammern () führen zu einer Zustandsänderung. Eckige Klammern [] beinhalten Aktionen, die beim Übergang ausgeführt werden. Die zwei senkrechten Linien || bedeuten Negation des eingeschlossenen Ausdrucks. Wie allgemein üblich steht »+«für ein logisches ODER und »*« für eine logische UND-Verknüpfung. Ein Pfeil »->« signalisiert einen Übergang in einen neuen Zustand.
Zum Abschluß noch eine Anmerkung: Bei den Versuchen, eine Direkt-Ausgabe zu programmieren, stand ich vor einigen Problemen. Vor allem erwartete ich nach dem Studium des Hardware-Manual, daß es sofort ausgegeben wird, sobald ein Zeichen in AUDxDAT geschrieben ist. Danach kehrt der Automat zwar in den Grundzustand zurück. Schreibt man aber laufend Werte in AUDxDAT, so sollte doch aus dem Lautsprechermonitor etwas zu hören sein, wenn auch keine besondere Qualität zu erwarten ist. Bei Experimenten stellte sich dann heraus, daß dies nicht der Fall ist. Der Übergang aus dem Grundzustand findet statt, sobald ein Wert in AUDxDAT geschrieben wurde und löst einen Interrupt aus. Wird dieser Interrupt nicht rechtzeitig oder gar nicht beantwortet, findet keine Ausgabe statt. Die Maschine kehrt unmittelbar in den Grundzustand zurück. Ist dies der Fall, so ist das Zustandsübergangsdiagramm durch einen Pfeil von 010 nach 000 zu ergänzen. Dieser Übergang ist aber in der Dokumentation von Commodore nicht erwähnt. Bei eigenen Experimenten ist es auf jeden Fall hilfreich zu wissen, daß ohne eine eigene Interrupt-Routine, die die Anforderungen von »Paula« bei der Interrupt-gesteuerten Ausgabe beantwortet, der Amiga zu keinem Piep veranlaßt werden kann (siehe auch 2.5 »Direkt-Ausgabe per Interrupt«).

Taken from Amiga-Manuals-Website

Systemsoftware – das Audio-Device 3

Das Amiga-Betriebssystem verwendet für die Durchführung jeglicher Ein- und Ausgabe (I/O : Input/Output) eine bestimmte Klasse von Softwaremodulen, die sogenannten Devices. Diese stellen für ein Amiga-Programm die Software-Schnittstelle zu jeder Ein-/Ausgabe-orientierten Hardware dar, wie z.B. Bildschirm, Tastatur, Massenspeicher oder Drucker. Auch das Amiga-DOS verwendet die Devices zur Kommunikation mit den jeweils angesprochenen Geräten. Die zutreffendste Übersetzung des Begriffs »Device« würde deshalb wohl »Gerätetreiber« lauten.

Auch für die Audio-Hardware des Amiga ist ein solches Device vorgesehen. Seine Benutzung empfiehlt sich vor allem dann, wenn Audio-Anwendungen innerhalb der Multitasking-Umgebung des Amiga verwendet werden, d.h. die Hardware unter Umständen von mehreren Tasks zugleich benutzt werden soll. Da die Verwendung eines Devices eine recht gute Kenntnis des Message- und I/O-Prinzips des Amiga verlangt, soll im folgenden Kapitel zuerst auf die Struktur und Programmierung von Devices und der zugehörigen Ein/Ausgabe-Techniken eingegangen werden. Hierbei kann selbstverständlich nur eine kurze Beschreibung geliefert werden, da eine ausführliche Diskussion dieser Themen den Rahmen dieses Kapitels sprengen würde. Für weitere Informationen siehe [Addison-Wesley: The ROM Kernel Manual: Exec / Addison-Wesley: The ROM Kernel Manual: Libraries & Devices].

Danach folgt die Besprechung der für das Audio-Device spezifischen Funktionen und Kommandos, wobei sowohl die Aspekte von direktem Hardwarezugriff als auch des Schreibzugriffs auf das Device (*CMD_WRITE*) erläutert werden sollen. Da sich wohl die meisten Programmierer, die hauptsächlich die Systemsoftware zur Unterstützung ihrer Programme verwenden, einer Hochsprache bedienen, orientieren sich die Angaben und Beispiele dieses Kapitels an der Programmiersprache C.

3.1 Was ist ein Device?

3.1.1 Messages und Ports

Um die Funktionsweise eines Device verstehen zu können, bedarf es einiger Kenntnisse über das Message-System des Amiga. Zwei Tasks, also zwei voneinander unabhängig laufende Programme, können im Amiga dadurch miteinander kommunizieren, indem sie sogenannte Messages (Botschaften) austauschen. Diese Messages müssen an Message-Ports übergeben werden, die den Listenkopf für die Liste der am Port angekommenen Messages enthalten. Grundsätzlich betrachtet ist eine Message lediglich ein Speicherbereich, der einem anderen Task für eine gewisse Zeit zum Lesen (oder Schreiben) zur Verfügung gestellt wird und während dieser Zeit natürlich auch vom sendenden Task nicht verändert werden darf. Hat der Empfänger-Task die

Message verarbeitet, so muß er dies dem Sender mitteilen, indem er die Message »beantwortet« (Reply). Jetzt kann der von der Message belegte Speicher wieder erneut verwendet oder an das System zurückgegeben werden.

Ein solcher Message-Port hat folgendes Aussehen:

```
struct MsgPort {
  struct Node mp_Node;
  UBYTE mp_Flags;
  UBYTE mp_SigBit;
  struct Task *mp_SigTask;
  struct List mp_MsgList;
};
```

mp_Node:
Node-Struktur, mit der der Port in die Liste der Message-Ports eingebunden ist.

mp_Flags:
Flags zur Bestimmung der Aktion, die auf das Eintreffen einer Message hin erfolgen soll: *PA_SIGNAL* – der in *mp_SigTask* angegebene Task erhält das in mp_SigBit angegebene Signal. *PA_SOFTINT* – der in *mp_SigTask* angegebene Software-Interrupt wird ausgelöst. *PA_IGNORE* – keine Aktion.

mp_SigBit:
Nummer des Signalbits, das bei Eintreffen einer Message gesendet wird.

mp_SigTask:
Zeiger auf den Task, an den das Signal gesendet wird bzw. auf eine IntVector-Struktur.

mp_MsgList:
Listenkopf für die Liste der eingetroffenen Messages.

Die Messages, die bei Eintreffen in diese Liste eingebunden werden, enthalten eine feste Grundstruktur für die Übergabe und Einbindung sowie einige beliebig lange Erweiterungen, die die für die Message spezifischen Zusatzinformationen enthalten. Diese Grundstruktur ist so definiert:

```
struct Message {
    struct Node mn_Node;
    struct MsgPort *mn_ReplyPort;
    UWORD mn_Length;
};
```

mn_Node:
Node-Struktur, mit der die Message in die Liste der am Port angekommenen Messages eingebunden wird.

Taken from Amiga-Manuals-Website

mn_ReplyPort:
Message-Port, an den die Message zur Beantwortung zurückgeschickt werden soll.

mn_Length:
Länge der Message in Bytes.

Möchte man eine Message an einen Task senden, kennt aber die Adresse des Empfängerports nicht, so kann man diese anhand des Namens ermitteln, auf den dieser Port im *ln_Name*-Feld seiner *mp_Node*-Struktur verweist. Dies geschieht mit Hilfe der *FindPort*-Routine:

```
struct MsgPort *port;

if(!(port = FindPort("port-name"))
   printf("Konnte Port nicht finden! \n");
```

Haben wir den Port gefunden, so müssen wir nur einige Strukturelemente der Message initialisieren, wie zum Beispiel den Antwortport. Da der Speicherbereich einer Message auf einer Langwort-Grenze beginnen muß, muß er jedoch vorher mit AllocMem reserviert werden. Nun können wir unsere Message senden, was mit der PutMsg-Routine erledigt wird:

```
struct MsgPort *port, *replyport;
struct XMessage *xmsg;   /* Message eines beliebigen Typs */

if(!(xmsg=(struct Message*)AllocMem(sizeof(Message),MEMF_PUBLIC))) {
   printf("Kein Speicher frei!");
   return();
}

if(!(replyport = CreatePort("ReplyPort-Name",0))) {
   printf("Konnte Port nicht herstellen!");
   FreeMem(xmsg,sizeof(Message));
   return();
}

xmsg -> x_Msg.mn_Node.ln_Type = NT_MESSAGE;
xmsg -> x_Msg.mn_ReplPort = replyport;

PutMsg(port,xmsg);
```

Der Empfängertask könnte nun beispielsweise auf das Eintreffen der Message an einem Port warten, indem er die Funktion *WaitPort()* aufruft. Dabei wird der Task »schlafengelegt«, d.h., während des Wartens verbraucht er keine Rechenzeit. Dazu müssen die Strukturelemente des Portsk *mp_Flags* mit dem Wert *PA_SIGNAL* und *mp_SigBit* mit der Nummer des Signals initialisiert sein, das bei Eintreffen der Message an den Task gesendet wird und ihn damit wieder zum Leben erweckt. Dies wird zwar im allgemeinen bereits von der C-Funktion *CreatePort()* erledigt (kein Bestandteil des Amiga-Betriebsystems!), hier sind die benötigten Schritte jeoch noch einmal im einzelnen:

```
struct MsgPort *port;
struct Message *msg, *WaitPort();
int SigBit;
```

```
if((Sigbit = AllocSignal(-1)) == -1) {
   printf("Kein Signal frei!");
   return();
}
if(!(port=(struct MsgPort *)AllocMem(sizeof(MsgPort),MEMF_PUBLIC))) {
   printf("Kein Speicher frei!");
   FreeSignal(SigBit);
   return();
}

port -> mp_Flags = PA_SIGNAL;
port -> mp_SigBit = SigBit;
port -> mp_SigTask = FindTask(0);    /* Dieser Task selbst */

msg = WaitPort(port);
```

WaitPort() gibt einen Zeiger auf die erste Message in der Liste zurück, ohne jedoch diese Message aus der Liste zu entfernen.

Jetzt muß sich der Empfängertask die empfangenen Messages aus dem Port abholen, was mit der *GetMsg()*-Routine geschieht. Diese entfernt die erste Message aus der Liste (also die zuerst angekommene) und gibt einen Zeiger auf diese zurück. Waren keine Messages in der Liste, so wird eine 0 zurückgegeben. Das folgende Beispiel holt sämtliche Messages von einem Port ab und leert ihn damit:

```
while(msg = GetMsg(port)) {
   printf("Message Adresse %ld gelesen!",msg);
}
```

Da die Messages immer nur in der zeitlichen Reihenfolge ihres Eintreffens geholt werden und damit solche mit einer hohen Dringlichkeit unter Umständen erst stark verspätet abgearbeitet werden könnten, empfiehlt es sich, solche wichtigen Messages über einen separaten Port zu empfangen.

Nun muß dem sendenden Task noch mitgeteilt werden, daß die Message gelesen wurde und sie selbst jetzt nicht mehr benötigt wird, damit ihr Speicher weiterverwendet werden kann. Dies geschieht normalerweise mit der *ReplyMsg()*-Funktion:

```
while(msg = GetMsg(port)) {
   printf("Message Adresse %ld beantwortet!",msg);
   ReplyMsg(msg);
}
```

Dieser Vorgang wird auch oft dazu benutzt, Ergebnisse an den Sender zurückzugeben. Dies wird meist erreicht, indem der Empfängertask vor dem Antworten die Daten der Message verändert und dann *ReplyMsg()* aufruft. Wenn dies geschehen ist, kann der Sender die so zurückerhaltenen Informationen in Ruhe auswerten, da der Speicherbereich der Message ja jetzt konventionsgemäß nur noch von ihm benutzt und verändert werden darf.

Tekkotronics-Manuals-Website

3.1.2 Device-I/O

Ein Device ist eine Sammlung wohldefinierter Interaktionen zwischen der aufrufenden Soft- und der Hardware. Es besteht im wesentlichen aus einer Device-Struktur, die in die Device-Liste des Systems eingebunden ist, einem Codemodul und einer Sprungtabelle auf die Funktionen des Device. Mehrere Einheiten, die mit demselben Device angesprochen werden (z.B. mehrere Diskettenlaufwerke, Festplattenpartitionen oder auch Audio-Kanäle), nennt man die Units dieses Devices. Da mit jedem der Units separat interagiert und Information ausgetauscht werden muß, existiert für jedes von ihnen mindestens eine sogenannte *I/O-Request*-Struktur, über die diese Kommunikation abgewickelt wird. Ein I/O-Request ist folgendermaßen definiert:

```
struct I/O-Request {
  struct Message io_Message;
  struct Device *io_Device;
  struct Unit *io_Unit;
  UWORD io_Command;
  UBYTE io_Flags;
  BYTE io_Error;
};
```

io_Message:
Message-Struktur, mit dem der I/O-Request in die Liste der bei einem Device anhängigen Requests eingebunden wird.

io_Device:
Zeiger auf die Device-Struktur. Dieser Zeiger wird vom System beim Aufruf von *OpenDevice()* initialisiert.

io_Unit:
Zeiger auf die Unit-Struktur, der dieser I/O-Request zugeordnet ist (sollte vom User nicht verändert werden), auch anderer Device-abhängiger Inhalt. Dieses Element wird vom System beim Aufruf von *OpenDevice()* initialisiert.

io_Command:
Nummer des Kommandos, das ausgeführt werden soll. Dies kann entweder ein Standard- (siehe unten) oder ein Device-spezifisches Kommando sein.

io_Flags:
Dient zur Angabe von I/O-Optionen und zum Abfragen verschiedener Statusinformationen.

io_Error:
Nummer eines eventuellen Fehlers nach Beendigung des I/O-Requests.

Da die Elemente *io_Device*, *io_Unit* und *io_Command* durch die Durchführung eines I/O-Vorgangs nicht beeinflußt werden, kann z.B. mit einer einzigen *I/O-Request*-Struktur mehrere Male der gleiche Befehl übergeben werden, ohne irgendeine Korrektur vornehmen zu müssen.

Tandon Amiga-Manuals

Diese Grundform der *I/O-Request*-Struktur wird noch durch die jeweiligen Device-spezifischen Elemente erweitert, die z.B. die Adresse und Länge der zu übergebenden Daten etc. enthalten. Die für das Audio-Device vorgesehene Variation des I/O-Requests wird im Abschnitt 3.2 erläutert.

Das Betriebssystem sieht vier Routinen zur Behandlung von I/O-Requests vor, die allerdings nur den I/O-Request-Struktur als ganzes an das Device übergeben, ohne dabei seinen Inhalt zu berücksichtigen:

DoIO:
Beginnt den I/O-Vorgang und wartet auf seine Beendigung (synchroner I/O).

SendIO:
Beginnt den I/O-Vorgang, ohne auf seine Beendigung zu warten (asynchroner I/O).

WaitIO:
Wartet auf die Beendigung eines laufenden I/O-Vorgangs.

CheckIO:
Überprüft, ob ein I/O-Vorgang bereits beendet ist.

Zusätzlich zu diesen sehr allgemeinen Funktionen gibt es auch noch eine Möglichkeit, die entsprechenden Routinen im Device direkt aufzurufen. Zwei Routinen erlauben sowohl den Start als auch den Abbruch eines I/O-Vorgangs, wobei die Verwendung dieser Routinen mit einer genaueren Kenntnis des Device verbunden und daher mit einiger Vorsicht zu genießen ist.

BeginIO:
Beginnt den I/O-Vorgang (synchroner oder asynchroner I/O, abhängig vom Device).

AbortIO:
Versucht, den gerade laufenden I/O-Vorgang abzubrechen.

Dasjenige Strukturelement, in dem dem Device die eigentlich durchzuführende Aktion mitgeteilt wird, ist *io_Command*. Das darin enthaltene Device-Kommando besitzt die folgenden Standard-Typen:

CMD_RESET:
Dies versetzt das gesamte Device in den Ausgangszustand, streicht alle anhängigen I/O-Vorgänge und initialisiert die jeweilige Hardware neu.

CMD_READ:
Startet einen Lesevorgang vom Device (siehe Device-spezifische I/O-Request-Strukturen).

CMD_WRITE:
Startet einen Schreibvorgang auf das Device.

CMD_CLEAR:
Löscht vorher eingeleitete I/O-Vorgänge.

CMD_STOP:
Unterbricht einen I/O-Vorgang.

CMD_START:
Setzt einen I/O-Vorgang fort nach *CMD_STOP*.

CMD_FLUSH:
Bricht alle anhängigen I/O-Vorgänge ab und beantwortet alle mit einer Fehlermeldung.

CMD_NONSTD:
Ab dieser Befehlsnummer beginnen die Device-spezifischen Kommandos, jeweils definiert durch *CMD_NONSTD+0*, *CMD_NONSTD+1* usw.

Alle vorhandenen Devices werden im Amiga nur anhand ihres Namens unterschieden. Um ein Device zu öffnen, muß man die Exec-Routine OpenDevice aufrufen:

OpenDevice(devicename, Unit, I/O-Request, flags) (A0, A1, D0, D1)

devicename:
Zeiger auf den Namen des Device (mit angehängter 0).

Unit:
Zeiger auf Unit-Struktur.

I/O-Request:
Zeiger auf I/O-Request.

flags:
Flags für I/O-Request.

Diese gibt einen Zeiger auf die Device-Struktur zurück, wenn der Aufruf erfolgreich war, und eine 0 im Fehlerfall.

3.2 Audioausgabe als Device

Wie alle Devices, so ist auch das Audio-Device für die Behandlung von mehreren gleichartigen Hardware-Einheiten vorgesehen, in diesem Fall der einzelnen Audio-Kanäle des Amiga. Da man in den meisten Anwendungen nicht nur einen Kanal ansprechen möchte, wurde eine Möglichkeit geschaffen, die meisten Kommandos des Audio-Device an mehrere Units, also Kanäle gleichzeitig zu senden. Aus diesem Grund enthält das *io_Unit*-Element des I/O-Requests auch keinen Zeiger auf ein Unit, sondern lediglich in den untersten 4 Bit die Kombination von Kanälen, die mit diesem I/O-Request angesprochen werden sollen. Die Zuteilung der einzelnen Kanäle an einen Aufrufer erfolgt mit Hilfe des *ADCMD_ALLOCATE*-Befehls, wobei die verschiedenen Kanalkombinationen, die für die jeweilige Anwendung akzeptabel wären, in einer Tabelle angegeben werden können. Diese Tabelle wird dann beim Aufruf von *ADCMD_ALLOCATE* nach einer (im Moment) freien Kanalkombination durchsucht. Hierbei spielt vor allem eine Rolle, welche Priorität bei diesem Aufruf angegeben wurde und mit welcher Priorität eventuell andere Kanäle bereits belegt wurden. Es sind Prioritätswerte von –128 bis 127 möglich, wobei 127 die höchste, also »unaufhaltbar« bedeutet (Näheres siehe *ADCMD_ALLOCATE*).

Tomas Amiga-Manuals-Website

Die für das Audio-Device vorgesehene Variation der *I/O-Request*-Struktur ist wie folgt definiert:

```
struct IOAudio {
struct I/O-Request ioa_Request;
WORD ioa_AllocKey;
UBYTE *ioa_Data;
ULONG ioa_Length;
UWORD ioa_Period;
UWORD ioa_Volume;
UWORD ioa_Cycles;
struct Message ioa_WriteMsg;
};
```

ioa_AllocKey:

Anhand dieses Schlüssels identifiziert das Device den »rechtmäßigen« Benutzer der angesprochenen Kanäle. Erfolgt ein unberechtigter Zugriff, so wird der Request mit einer Fehlermeldung beantwortet.

ioa_Data:
Zeiger auf die abzuspielende Wellenform im Chip-RAM (bei *CMD_WRITE)* oder auf die Tabelle der Kanalkombinationen *(ADCMD_ALLOCATE)*.

ioa_Length:
Länge der Wellenform in Bytes (gerade Anzahl).

ioa_Period:
Tonhöhe (Periodenzählerwert).

ioa_Volume:
Lautstärke.

ioa_Cycles:
Anzahl der zu spielenden Zyklen der Wellenform (0 = kontinuierlich).

ioa_WriteMsg:
nur interne Verwendung (???).

Das *ioa_AllocKey*-Element spielt bei der Verwendung von mehreren I/O-Requests eine besondere Rolle: Möchte man für eine mit *ADCMD_ALLOCATE* belegte Kanalkombination von einem anderen I/O-Request aus ansteuern, so muß man sowohl das *io_Unit* – als auch das *ioa_Allockey*-Element des Requests, der die Kanäle besetzt hat, in den neuen Request kopieren. Damit wird diesem die Erlaubnis gegeben, die Kanäle zu benutzen, andernfalls wird jeder I/O-Zugriff mit einer *ADIO_NOALLOCATION*-Fehlermeldung quittiert.

Für die Abwicklung der meisten Kommandos empfiehlt sich die Verwendung der Routine *BeginIO*, da die Routinen *DoIO* oder *SendIO* den Inhalt des io_Flags-Feldes verändern, was z.B. bei *CMD_WRITE* zu Schwierigkeiten führen kann. Lediglich Befehle, grundsätzlich eine längere

Tappan Amiga-Manuals-Website

Zeit nicht beantwortet werden (wie z.B. *ADCMD_LOCK*) können mit *SendIO* abgeschickt werden.

Für das Audio-Device stehen also folgende Routinen zur Verfügung:

OpenDevice():
Öffnet das Device und fügt es in die Device-List ein. Wird diese Routine mit einem anderen Wert als 0 im *ioa_Length*-Element aufgerufen, so wird die in *ioa_Data* angegebene Kanaltabelle auf eine mögliche Kombination durchsucht (siehe *ADCMD_ALLOCATE*). Dies wird mit gesetztem *ADIO_NOWAIT*-Flag durchgeführt, so daß der Belegungsversuch beim Scheitern sofort abgebrochen wird. Ist keine Kanalbelegung erwünscht, so muß *ioa_Length* eine 0 enthalten.

CloseDevice():
Schließt das Device und führt ein ADCMD_FREE-Kommando bezüglich der in der *io_Unit*-Struktur des angegebenen Requests angegebene Kanäle durch. Sind jedoch mehrere ADCMD_ALLOCATE-Befehle durchgeführt worden und deshalb verschiedene *ioa_AllocKey*-Werte in den einzelnen Requests vorhanden, so müssen die von ihnen belegten Kanäle zuerst separat mit ADCMD_FREE freigegeben werden. Erst dann darf CloseDevice aufgerufen werden.

BeginIO():
Startet einen I/O-Vorgang. Unterscheidet sich von den Routinen der anderen Devices nur in der Tatsache, daß lediglich ein Zeiger auf den I/O-Request als einziger Parameter benötigt wird.

AbortIO():
Kann benutzt werden, um die Befehle *ADCMD_LOCK*, *ADCMD_ALLOCATE*, *CMD_WRITE* und *ADCMD_WAITCYCLE* abzubrechen. Beim Audio-Device ist der Aufruf dieser Routine immer erfolgreich.

Die im Audio-Device vorgesehenen Kommandos lassen sich grob in zwei Gruppen einteilen: Die Kommandos zur Zuteilung und Belegung von Audio-Kanälen und die Hardware-orientierten Kommandos. Erstere besorgen die Steuerung der I/O-Vorgänge zur Kanalbelegung:

ADCMD_ALLOCATE

Durch dieses Kommando wird der Zugriff auf eine Kanalkombination ermöglicht. Bei Aufruf dieses Kommandos muß im *ioa_Data*-Element ein Zeiger auf und in *ioa_Length* die Länge einer Tabelle angegeben werden, die die möglichen erwünschten Kanalkombinationen enthält. Die Tabelle besteht aus einzelnen Byte-Werten, deren unterste 4 Bit in das *io_Unit*-Element des Requests übernommen werden, wenn eine Kombination belegt werden konnte. Die Nummer des Bits steht dabei für die Kanalnummer, also 0 und 3 für den linken, 1 und 2 für den rechten Kanal.

Die Abarbeitung der Tabelle wird dabei wie folgt durchgeführt: Zuerst wird versucht, die erste Kanalkombination in der Tabelle zu belegen. Ist der Versuch erfolgreich, so wird die Kombination in das *io_Unit*-Element und der jeweilige Key in das *ioa_AllocKey*-Element des I/O-Requests eingetragen. Gelingt dies aber nicht, da einer der angeforderten Kanäle besetzt ist (egal, mit welcher Priorität), so wird dasselbe mit der zweiten Kombination in der Tabelle durchgeführt, anschließend mit der dritten usw. Auf diese Weise wird die gesamte Tabelle durchsucht, bis eine freie Kombination gefunden ist.

War dieser Durchgang erfolglos, so wird die Tabelle noch ein zweitesmal durchsucht, diesmal nach einer Kombination, in der die angeforderten, aber bereits besetzten Kanäle alle eine niedrigere Priorität aufweisen als die unseres I/O-Requests (*ln_Pri*-Feld der *io_Message*-Struktur) und damit »gestohlen« werden dürfen. Auch hier spielt die Reihenfolge der Kombinationen eine Rolle, da die am Beginn der Tabelle natürlich zuerst ausprobiert werden und ihnen damit der Vorzug vor allen anderen Möglichkeiten gegeben wird.

Ist es aber nicht möglich, eine der Kombinationen in der Tabelle zu besetzen, ohne dabei einen Kanal mit gleicher oder höherer Priorität stehlen zu müssen, so wartet das Device entweder auf das Freiwerden einer Kombination oder es wird eine Fehlermeldung zurückgegeben. Dies ist abhängig davon, ob das *ADIOF_NOWAIT*-Bit im *io_Flags*-Feld beim Beginn des I/O-Vorgangs gesetzt war. Stand dieses Bit auf 0, so wird der Request vorerst nicht beantwortet, sondern von nun an bei jedem Aufruf von *ADCMD_FREE* überprüft, ob eine der Kanalkombinationen der Tabelle nun frei ist. Dieser Wartezustand kann z.B. mit *AbortIO()* abgebrochen werden. War das *ADIOF_NOWAIT*-Bit jedoch auf 1 gesetzt, so überprüft das Device die Tabelle nur einmal und gibt im Falle eines Mißerfolgs die Fehlermeldung *ADIOERR_ALLOCFAILED* zurück.

Hier ein paar Beispiele für die Verwendung des *ADCMD_ALLOCATE*-Kommandos:

Zuerst einmal müssen wir die Tabelle der Kanalkombinationen erstellen, die unseren Ansprüchen genügen. Nehmen wir z.B. an, wir möchten einen Ton auf den beiden Stereoausgängen wiedergeben, wobei ja für den linken Ausgang die Kanäle 0 u. 3 und für den rechten 1 u. 2 zur Verfügung stehen. Eine entsprechende Tabelle würde also so aussehen:

```
#define CH0B_LEFT  0
#define CH1B_RIGHT 1
#define CH2B_RIGHT 2
#define CH3B_LEFT  3
#define CH0F_LEFT  1
#define CH1F_RIGHT 2
#define CH2F_RIGHT 4
#define CH3F_LEFT  8
UBYTE AllocationTable[] = {
   CH0F_LEFT | CH1F_RIGHT,  /* Kanal 0 und 1 */
   CH0F_LEFT | CH2F_RIGHT,  /* Kanal 0 und 2 */
   CH3F_LEFT | CH1F_RIGHT,  /* Kanal 3 und 1 */
   CH3F_LEFT | CH2F_RIGHT   /* Kanal 3 und 2 */
};
```

Oder betrachten wir den Fall, daß zwei Kanäle benötigt werden, die einander modulieren sollen. Da ein Kanal nur den mit der nächsthöheren Nummer modulieren kann (siehe Kapitel 2.2.4), muß die zugehörige Tabelle so lauten (die bevorzugte Kombination sollte man natürlich an den Anfang stellen):

```
UBYTE AllocationTable[] = {
   CH0F_LEFT | CH1F_RIGHT,       /* Kanal 0 und 1 */
   CH1F_RIGHT | CH2F_RIGHT,      /* Kanal 1 und 2 */
   CH2F_RIGHT | CH3F_LEFT,       /* Kanal 2 und 3 */
};
```

In Begleitung dieses Kommandos müssen die folgenden Parameter übergeben werden: Erstens muß die Adresse der Tabelle in das *ioa_Data*-Feld des Requests geschrieben werden, zweitens muß das *ioa_Length*-Element deren Länge in Bytes enthalten. Dies kann z.B. durch folgende Zeilen geschehen:

```
struct IOAudio *ioaudio

ioaudio -> ioa_Data = AllocationTable;
ioaudio -> ioa-Length = sizeof(AllocationTable);
```

ADCMD_FREE

Mit diesem Kommando können Kanäle, die vorher mit *ADCMD_ALLOCATE* belegt wurden, wieder freigegeben werden. Dies ist vor allem dann nötig, wenn mehrere Male Kanäle angefordert wurden und damit verschiedene *AllocKey*-Werte existieren, so daß nicht mehr alle Kanäle auf einmal durch *CloseDevice()* freigegeben werden können. Durch *ADCMD_FREE* werden die im *io_Unit*-Element angegebenen Kanäle zurückgesetzt (d.h. ein *ADCMD_RESET*-Befehl wird ausgeführt) und vom Device zur weiteren Benutzung freigegeben. Kanäle, die dem Task »gestohlen« werden, brauchen nicht wieder freigegeben zu werden, das obliegt in diesem Fall dem »Dieb«.

ADCMD_LOCK

Mit Hilfe dieses Kommandos läßt sich überprüfen, ob einer der dem Request zugeteilten Kanäle von einem anderen (mit höherer Priorität) gestohlen wurde. Führt man das *ADCMD_LOCK*-Kommando aus, so wird der Request so lange nicht beantwortet, bis einer der im *io_Unit*-Element angegebenen Kanäle gestohlen wird. Dadurch ist es im allgemeinen notwendig, für dieses Kommando einen eigenen I/O-Request plus *ReplyPort()* zu verwenden, der lediglich auf dessen Beantwortung wartet. Aus Sicherheitsgründen sollte dies bei jeder Anwendung der Audio-Hardware so gehandhabt werden, um ein Programm auf das Stehlen seiner Kanäle schnell genug reagieren zu lassen und damit Konflikte zu vermeiden.

Das *ADCMD_LOCK*-Kommando wird folgendermaßen verwendet: Zuerst werden die gewünschten Kanäle mit *OpenDevice()* oder *ADCMD_ALLOCATE* belegt. Dann wird mit demselben Request oder einem anderen, in den die entsprechenden *io_Unit*- und *ioa_AllocKey*-Werte kopiert wurden, der *ADCMD_LOCK*-Befehl durchgeführt. Jetzt kann die Hardware der belegten Kanäle frei nach Wunsch manipuliert werden. (Bei der Verwendung von *CMD_WRITE* ist *ADCMD_LOCK* nicht nötig, da das Device alle nötigen Schritte zur Rücksetzung hier selbständig durchführt.) Verläuft alles reibungslos, so wird der Benutzer der Kanäle diese nach Gebrauch durch *ADCMD_FREE* wieder ordnungsgemäß freigeben, worauf der *ADCMD_LOCK*-Request beantwortet wird. Wird jedoch während der Benutzung einer der Kanäle gestohlen, so wird der *ADCMD_LOCK*-Request sofort beantwortet, und zwar mit der Fehlermeldung *ADIOERR_CHANNELSTOLEN*. Jetzt müssen die belegten Kanäle so schnell wie möglich freigegeben werden, wobei das Programm natürlich sämtliche auf diese Kanäle bezogenen Aktionen (wie z.B. Interrrupts) stoppen muß. Dann muß noch ein *ADCMD_FREE*-Befehl ausgeführt werden, um die Kontrolle über die Kanäle an den »Dieb« zu übergeben. Will das

Programm die gestohlenen Kanäle so bald wie möglich wiederhaben, so sollte es noch einmal mit *ADCMD_ALLOCATE* versuchen, sie zu belegen, wobei dann natürlich das *ADIOF_NOWAIT*-Bit in *io_Flags* auf 0 stehen muß.

Vom Standpunkt eines Programms, das einen Kanal übernehmen möchte, stellt sich das so dar: Zuerst wird versucht, die Kanalkombination mit *ADCMD_ALLOCATE* zu belegen. Ist einer der Kanäle bereits mit *ADCMD_LOCK* »gelockt«, so wird mit der Beantwortung des Requests jedoch so lange gewartet, bis der Kanal durch *ADCMD_FREE* freigegeben wurde, und zwar unabhängig davon, ob dabei das *ADIOF_NOWAIT*-Flag gesetzt war oder nicht. Erst wenn der Request beantwortet wird, ist der übernehmende Request Herr über die angeforderten Kanäle. Dies macht deutlich, daß der bestohlene Request seine Kanäle unbedingt immer sofort freigeben muß, da sonst andere Aufrufer hörbar behindert würden.

ADCMD_SETPREC

Möchte man die Priorität einer einmal angeforderten Kanalkombination ändern, so kann das mit Hilfe von *ADCMD_SETPREC* geschehen. Hat man eine Meldung an den User, wie z.B. die Beendigung eines Berechnungsvorgangs oder einen Zeitgong, so ist meist nur der Beginn des Klanges notwendig, um die gewünschte Meldung mitzuteilen, der Rest soll jedoch andere wichtigere Audioanwendungen nicht übertönen. Dies kann dadurch erreicht werden, indem zuerst der *ADCMD_ALLOCATE*-Befehl mit einer entsprechend hohen Priorität durchgeführt und der Beginn der Klangfolge ausgegeben wird, dann aber mit *ADCMD_SETPREC* auf eine niedrigere Priorität umgeschaltet wird und erst jetzt die restlichen Töne ausgegeben werden. Dies gewährleistet eine optimale zeitliche Verteilung der verfügbaren Kanäle.

Die folgenden Kommandos sind für die Klangausgabe selbst zuständig und beeinflussen deshalb die Hardware direkt:

CMD_WRITE

Dieses Kommando ermöglicht die Ausgabe einer im Chip-RAM abgelegten Wellenform, wobei sowohl eine bestimmte Anzahl von Wiederholungen als auch eine konstante Wiedergabe gewählt werden können. Es kann jeweils nur auf einen Kanal angewendet werden. Sind fälschlicherweise mehrere Bits in *io_Unit* gesetzt, so löst das nicht etwa einen Fehler aus, sondern das unterste Bit wird hergenommen und alle anderen werden gelöscht. Die Elemente des *IOAudio*-Requests haben hier die folgenden Bedeutungen:

ioa_Data:
Zeiger auf die Wellenform im CHIP-RAM (muß auf einer geraden Adresse beginnen).

ioa_Length:
Länge der Wellenform in Bytes (muß eine gerade Anzahl sein).

ioa_Period:
Wenn das ADIOF_PERVOL-Flag gesetzt ist, wird der hier angegebene Periodenwert übernommen, ansonsten wird der zuletzt bestimmte verwendet.

ioa_Volume:
Wenn das ADIOF_PERVOL-Flag gesetzt ist, wird der hier angegebene Lautstärkewert übernommen, ansonsten wird der zuletzt bestimmte verwendet.

ioa_Cycles:
Anzahl der Wiederholungen der Wellenform (0 = kontinuierliche Wiedergabe, bis AbortIO()).

ioa_WriteMsg:
Wenn das ADIOF_WRITEMESSAGE-Flag gesetzt ist, wird diese Message an das Device übergeben und beim Beginn der Abarbeitung des Requests beantwortet.

CMD_WRITE setzt den Datenpointer der DMA direkt auf die angegebene Wellenform. Deshalb muß diese erstens im CHIP-RAM auf einer Wortgrenze liegen und eine gerade Anzahl von Bytes haben, zweitens wird durch jede Änderung der Daten in der Wellenform der Klang direkt beeinflußt. Diese Tatsache kann zur Erzeugung spezieller Effekte ausgenützt werden.

Um die Werte für Lautstärke und Frequenz des Tones anzugeben, müssen nicht nur die Elemente *ioa_Period* und *ioa_Volume* korrekt initialisiert werden, sondern es muß auch das *ADIOF_PERVOL*-Flag gesetzt sein. Ist dies nicht der Fall, so werden für diesen Ton die letzten eingestellten Werte angenommen. Daraus folgt, daß zumindest der erste *CMD_WRITE*-Request mit gesetztem *ADIOF_PERVOL*-Flag durchgeführt werden muß, um überhaupt einen Ton zu erhalten.

Eine Möglichkeit zur Synchronisation mit dem Beginn der Tonausgabe bietet das *ADIOF_WRITEMESSAGE*-Flag. Wenn dieses Flag gesetzt ist, wird das *ioa_WriteMsg*-Feld als eine Message behandelt, die mit dem Request zum Device gesendet wird. Sie wird genau dann beantwortet, wenn die Ausgabe des im Request beschriebenen Tones beginnt, also die jeweiligen Werte in die Hardwareregister übernommen wurden. Dann kann ein Programm auf ein Signal hin oder innerhalb eines Softinterrupts entsprechend reagieren. Dies ermöglicht die rechtzeitige Vorbereitung der nächsten auszugebenden Wellenform und damit fließende Übergänge.

Eine der Anwendungen dieses Features ist das sog. Double-Buffering, also die Verwendung von zwei (oder mehr) CHIP-RAM-Bereichen, von denen jeweils der eine wiedergegeben und der andere auf die nächste Wiedergabe vorbereitet wird. Dabei muß im allgemeinen so vorgegangen werden (siehe auch Beispielprogramm PlayFile im Kapitel 2.2.5):

1. Versorge Puffer A mit Daten
2. Gib Befehl zur Ausgabe von Puffer A (mit *CMD_WRITE)*
3. Warte auf den Beginn der Wiedergabe von Puffer A
4. Versorge Puffer B mit Daten
5. Gib Befehl zur Ausgabe von Puffer B (mit *CMD_WRITE)*
6. Warte auf den Beginn der Wiedergabe von Puffer B
7. Gehe zurück zu Schritt 1, bis das Ende der Wiedergabe erreicht ist
8. Warte auf die Beendigung der Wiedergabe von Puffer B
9. Räum auf *(ADCMD_FREE)*!

Dies läßt sich allerdings auch ohne *ioa_WriteMsg* erreichen, indem man das Warten auf den Beginn des nächsten Ausgabevorgangs durch das Warten auf die Beendigung des letzten

austauscht. Wenn man für die Erledigung einer solchen Aufgabe nicht gleich einen eigenen (oder gar den eigenen) Task beschäftigen möchte, sollte man statt eines Signals einen Softinterrupt verwenden, indem man im *ioa_WritMsg*-Feld das *PA_SOFTINT*-Flag setzt und *mn_Replyport* auf die Interruptstruktur stellt, in der die Interruptroutine angegeben ist.

CMD_READ

Dieses Kommando ermöglicht die Bestimmung des momentan bearbeiteten *CMD_WRITE*-Kommandos. Es kann jeweils nur auf einen Kanal angewendet werden. Gibt man das *CMD_READ*-Kommando, so befindet sich nach Beantwortung des Requests ein Zeiger auf den gerade abgespielten Request im *ioa_Data*-Element.

CMD_STOP

Dieses Kommando hält die Ausgabe der in *io_Unit* angegebenen Kanäle an. Dies veranlaßt das Device, alle an diese Kanäle gegebenen *CMD_WRITE*-Befehle aufzustapeln und erst mit dem *CMD_START*-Befehl wieder mit der Ausgabe zu beginnen.

CMD_START

Dieses Kommando setzt die Wiedergabe der mit *CMD_STOP* angehaltenen Kanäle fort. Die Wiedergabe wird jedoch erst beim jeweils nächsten Wiedergabezyklus fortgesetzt, der Rest des vorhergehenden Zyklus kann nicht mehr berücksichtigt werden, da die Hardware ja die Bestimmung des aktuellen Pointers in der Wellenform nicht zuläßt (siehe Kapitel 2).

CMD_FLUSH

Dieses Kommando bricht alle *CMD_WRITE* und *ADCMD_WAITCYCLE*-Kommandos etc. der in *io_Unit* angegebenen Kanäle ab, indem es *AbortIO()* aufruft. *ADCMD_LOCK*-Kommandos können damit nicht rückgängig gemacht werden, das ist nur mit *ADCMD_FREE* möglich.

CMD_RESET

Dieses Kommando führt ein *CMD_FLUSH*-Kommando aus, setzt die Hardware der angegebenen Kanäle zurück (wie z.B. das *ADKCON*-Register etc.) und sorgt für die Wiederherstellung der alten Audio-Interrupts (wenn sie geändert wurden). *ADCMD_LOCK*-Kommandos können damit nicht rückgängig gemacht werden, das ist nur mit *ADCMD_FREE* möglich.

ADCMD_FINISH

Dieses Kommando bricht den jeweils gerade bearbeiteten *CMD_WRITE*-Befehl ab. Wie bei den meisten Kommandos, kann hierbei durch das Setzen des *ADIOF_SYNCCYCLE*-Flags auf das Ende des momentanen Zyklus gewartet werden. Dies spielt dann eine Rolle, wenn man damit zur Vermeidung von Klick-Geräuschen auf den nächsten Nulldurchgang des Audiowandlers warten will. (Natürlich müssen die verwendeten Wellenformen dann auch auf 0 beginnen und enden.)

ADCMD_PERVOL

Dieses Kommando verändert die Frequenz und Lautstärke des gerade in Bearbeitung befindlichen *CMD_WRITE*-Kommandos. Dies kann zur Erzeugung von Tremolos, Vibratos oder Hüllkurven eingesetzt werden. Durch das Setzen des *ADIOF_SYNCCYCLE*-Flags findet die Änderung erst beim Beginn des nächsten Zyklus statt.

ADCMD_WAITCYCLE

Dieses Kommando dient zur Synchronisation des Programmablaufs mit der Ausgabe der Wellenform. Das *ADCMD_WAITCYCLE*-Kommando wird genau dann beantwortet, wenn das Ende der Wellenform erreicht ist und ein neuer Ausgabezyklus beginnt. Wenn kein *CMD_WRITE*-Befehl anliegt, wird es sofort beantwortet.

Hier noch einmal eine Zusammenfassung aller Flags und Fehlermeldungen des Audio-Device:

IOF_QUICK:
1 = Request soll nicht beantwortet werden.

ADIOF_NOWAIT:
1 = *ADCMD_ALLOCATE* soll nicht auf das Freiwerden der Kanäle warten.

ADIOF_PERVOL:
1 = Übernehme *ioa_Period* und *ioa_Volume*-Werte des Requests, 0 = verwende zuletzt eingestellte Werte.

ADIOF_SYNCCYCLE:
1 = Warte mit der Bearbeitung dieses Requests auf das Ende des laufenden Wellenformzyklus.

ADIOF_WRITEMESSAGE:
1 = Beantworte Message in *ioa_WriteMsg* zu Beginn der Abarbeitung des Requests.

Fehlermeldungen:

ADIOERR_ALLOCFAILED:
Kanalbelegung durch *OpenDevice()* oder *ADCM_ALLOCATE* war erfolglos.

ADIOERR_NOALLOCATION:
Wert in *ioa_AllocKey* stimmt nicht mit den Keys der angesprochenen Kanäle überein.

ADIOERR_CHANNELSTOLEN:
Fehler, mit dem ein *ADCMD_LOCK*-Befehl beantwortet wird, wenn ein Kanal gestohlen wurde.

3.3 Ein Beispielprogramm zum Audio-Device

Das folgende Programm gibt eine Folge von Tönen über das Audio-Device aus, wie es etwa für ein akustisches Signal eines Anwenderprogramms typisch wäre. Zur Überwindung der dabei auftretenden Probleme wie z.B. das dynamische Allocieren von Requests wurde dabei ein besonderer Weg beschritten:

Text von Amiga-Manuals.de

```
/*****************************************************************************

    devplaynotes.c     (c) 6.4.1989 3-24 Nicolaus Wirsing

    All Rights Reserved

*****************************************************************************

    Dieses Programm holt sich die Werte für Peroide und Tondauer aus
    einer Tabelle (struct note[]) und übergibt sie mittels dynamisch
    allocierter IOAudio-Requests an das Audio-Device. Das
    Hauptproblem dabei ist jedoch, die Speicherbereiche der Requests
    möglichst bald wieder zurückzugeben, ohne dabei von einem eigenen
    Task aus ständig Aufräumaktionen durchführen zu müssen. Da von
    einem Interrupt aus keine speicherrelevanten Aktionen
    durchgeführt werden dürfen, muß man sich eines Tricks bedienen:
    Die Beantwortung eines Requests löst einen Soft-Interrupt aus,
    in dem der Zeiger auf die Exec-Routine Switch(), die für das
    Task-Switching zuständig ist, auf eine eigene Routine verbogen
    wird. In dieser können dann unbekümmert AllocMem() und Freemem()
    etc. aufgerufen werden, da Switch() nur dann angesprungen wird,
    wenn das Task-Switching explizit erlaubt ist. Danach muß noch der
    alte Zeiger auf Switch() in der Exec-Library wiederhergestellt
    werden. Dies verschafft eine einfache Möglichkeit, Speicherroutinen
    indirekt vom einem Interrupt aus aufzurufen, wobei man sich der
    baldmöglichsten Abarbeitung derselben sicher sein kann.

    ERRATA:

302:    jsr     _LVODisable(a6)         ; alle Interrupts verbieten

322:    jsr     _LVOEnable(a6)

    Wegen IntRoutine() muß dieses Programm mit absoluter Adressierung
    kompiliert werden, z.B. bei Aztec C:

    cc +L +D devplaynotes.c
    ln -o devplaynotes -lcl32 -lml32

***/


#include <exec/types.h>
#include <exec/ports.h>
#include <exec/memory.h>
#include <exec/interrupts.h>
#include <libraries/dos.h>
#include <devices/audio.h>

#define MAXVOLUME     64
#define PRECEDENCE    0
```

```
#define CH0B_LEFT    0
#define CH1B_RIGHT   1
#define CH2B_RIGHT   2
#define CH3B_LEFT    3
#define CH0F_LEFT    1
#define CH1F_RIGHT   2
#define CH2F_RIGHT   4
#define CH3F_LEFT    8


UBYTE AllocationTable[] = {
    CH0F_LEFT,
    CH1F_RIGHT,
    CH2F_RIGHT,
    CH3F_LEFT
};

BYTE waveForm[] = { 0, 49, 90, 117, 127, 117, 90, 49,
                    0,-49,-90,-117,-127,-117,-90,-49
                  };

#define A1   508    /*** Period-Werte der einzelnen Töne ***/
#define Ais1 480
#define B1   453
#define C1   428
#define Cis1 404
#define D1   381
#define Dis1 360
#define E1   339
#define F1   320
#define Fis1 302
#define G1   285
#define Gis1 269

#define A2   254
#define Ais2 240
#define B2   226
#define C2   214
#define Cis2 202
#define D2   190
#define Dis2 180
#define E2   170
#define F2   160
#define Fis2 151
#define G2   143
#define Gis2 135

            /*** Die einfachste Definition einer Note: Tonhöhe (PeriodenWert)
                 und Tondauer (hier: 1 = 1 sec, 1/2 = 1/2 sec etc.) ***/
struct note {
    int period;
    float duration;
};
```

Taken from Amiga-Manuals-Website

```
        /*** Ein Song besteht hier aus einem Vektor von Note-Strukturen ***/
struct note song1[] = {
    C1,(float)1/8,
    D1,(float)1/8,
    E1,(float)1/8,
    G1,(float)1/8,
    C2,(float)1/8,
    G1,(float)1/8,
    E1,(float)1/8,
    C1,(float)1/8,
    0,0
};

struct IOAudio    *allocIOA  = 0;
struct Device     *device    = 0;
struct MsgPort    *allocPort = 0, *soundPort = 0;
struct Interrupt  *softInterrupt = 0;

BYTE *waveData = 0;

BOOL listEmpty = TRUE;          /*** Bestimmt, ob SetFunction() nötig ist ***/

void (*OldSwitch)();

int IntRoutine();
int MySwitch();


main()
{
    int i;
    struct Message *msg;


                                /*** CHIP-RAM für Audiodaten reservieren ***/
    if(!(waveData =
                    (BYTE *)AllocMem(sizeof(waveForm), MEMF_PUBLIC | MEMF_CHIP) ))
    cleanup("\nKein Speicher frei !");

                                                /*** Audiodaten kopieren ***/
    for( i=0; i<sizeof(waveForm) ; i++)
    waveData[i] = waveForm[i];

    /*** Request für ADCMD_ALLOCATE reservieren ***/

    if(!(allocIOA = (struct IOAudio *)
                    AllocMem(sizeof(struct IOAudio),MEMF_PUBLIC|MEMF_CLEAR) ))
    cleanup("\nKein Speicher frei !");

                                                    /*** Device öffnen ***/
    if(OpenDevice(AUDIONAME,0,allocIOA,0) != 0)
    cleanup("\nKonnte Device nicht öffnen !");
```

Taken from Amiga-Manuals-Website

```
    device = allocIOA->ioa_Request.io_Device;

                                    /*** Port für ADCMD_ALLOCATE erstellen ***/
    if(!(allocPort = CreatePort("Audio-Beispiel",0)) )
    cleanup("\nKonnte Port nicht erstellen !");

                            /*** Request für ADCMD_ALLOCATE vorbereiten ***/
    allocIOA->ioa_Request.io_Message.mn_ReplyPort    = allocPort;
    allocIOA->ioa_Request.io_Message.mn_Node.ln_Pri = PRECEDENCE;
    allocIOA->ioa_Request.io_Command = ADCMD_ALLOCATE;
    allocIOA->ioa_Request.io_Flags   = ADIOF_NOWAIT;
    allocIOA->ioa_Data    = AllocationTable;
    allocIOA->ioa_Length = sizeof(AllocationTable);

    BeginIO(allocIOA);

    if(WaitIO(allocIOA))
    cleanup("\nKonnte Kanäle nicht belegen !");

    msg = GetMsg(allocPort);

                           /*** ReplyPort für SoftInterrupt vorbereiten ***/
    if(!(softInterrupt = (struct Interrupt *)
                AllocMem(sizeof(struct Interrupt),MEMF_PUBLIC|MEMF_CLEAR) ))
    cleanup("\nKein Speicher frei !");

    softInterrupt->is_Code = (void (*)())IntRoutine;

    if(!(soundPort = (struct MsgPort *)
    AllocMem(sizeof(struct MsgPort),MEMF_PUBLIC|MEMF_CLEAR) ))
    cleanup("\nKein Speicher frei !");

    soundPort->mp_Flags         = PA_SOFTINT;
    soundPort->mp_SigTask       = (struct Task *)softInterrupt;
    soundPort->mp_Node.ln_Type = NT_MSGPORT;

    NewList(&(soundPort->mp_MsgList));

                                        /*** Requests für Song abschicken ***/
    SendNotes(song1);

    printf("\nBitte CNTRL-C (^C) drücken!");

    Wait(SIGBREAKF_CTRL_C);
    cleanup("\nCTRL-C");
}

cleanup(string)
char *string;
{
    puts(string);
```

Taken from Amiga-Manuals-Website

```
    if(device)
    CloseDevice(allocIOA);

    if(waveData)
    FreeMem(waveData, sizeof(waveForm));

    if(allocIOA)
    FreeMem(allocIOA, sizeof(struct IOAudio));

    if(allocPort)
    DeletePort(allocPort);

    if(softInterrupt)
    FreeMem(softInterrupt, sizeof(softInterrupt));

    if(soundPort)
    FreeMem(soundPort, sizeof(struct MsgPort));

    exit(0);
}

SendNotes(song)
struct note *song;
{
    struct note *npoint;
    struct IOAudio *dynIOA;
    int cycles;


    if(song)
   {
       for(npoint=song;npoint->period;npoint++)
      {                                                    /*** 0 = Ende ***/

          if(!(dynIOA = (struct IOAudio *)AllocMem(sizeof(struct IOAudio),
                                                  MEMF_PUBLIC|MEMF_CLEAR) ))
          cleanup("\nKein Speicher frei !");

          dynIOA->ioa_Request.io_Message.mn_ReplyPort  = soundPort;
          dynIOA->ioa_Request.io_Device                = device;

          dynIOA->ioa_Request.io_Unit       = allocIOA->ioa_Request.io_Unit;
          dynIOA->ioa_Request.io_Flags      = ADIOF_PERVOL;
          dynIOA->ioa_Request.io_Command    = CMD_WRITE;
          dynIOA->ioa_AllocKey              = allocIOA->ioa_AllocKey;
          dynIOA->ioa_Data                  = (UBYTE *)waveData;
          dynIOA->ioa_Length                = sizeof(waveForm);
          dynIOA->ioa_Volume                = MAXVOLUME;
          dynIOA->ioa_Period                = npoint->period;
```

Takamon Amiga-Manuals-Website

```
            cycles = (int) (3570000 * npoint->duration /
                                        (sizeof(waveForm) * npoint->period) );
            dynIOA->ioa_Cycles = cycles;

            BeginIO(dynIOA);
        }
    }
}


            /*** In der Interruptroutine wird der Library-Eintrag für Switch
                                                    auf MySwitch gestellt ***/
IntRoutine()
{
    if(listEmpty)
   {
        listEmpty = FALSE;
        Forbid();
        OldSwitch = (void(*)())SetFunction( *((struct Library **)4),
                                                    -54L, MySwitch);
        Permit();
    }
}


#asm

public   _MySwitch

reserve  EQU     4                    ; für exec_lib.i
vsize    EQU     6
count    SET     -vsize*(reserve+1)
FUNCDEF  MACRO
_LVO\1   EQU     count
count    SET     count-vsize
ENDM

    include 'exec/types.i'
    include 'exec/exec_lib.i'
    include 'devices/audio.i'

_MySwitch:
    movem.l   d0-a6,-(a7)
    movea.l   4,a6
    jsr       _LVODisable(a6)            ; alle Interrupts verbieten

    movea.l   a6,a1
    movea.l   #-54,a0
    move.l    OldSwitch,d0
    jsr       _LVOSetFunction(a6)        ; alter Switch()-Zeiger

    movea.l   _soundPort,a2
```

Taken from Amiga-Manuals-Website

```
MSLp1:
   movea.l    a2,a0                          ; alle Requests zurückgeben
    jsr       _LVOGetMsg(a6)                 ; die bisher beantwortet wurden
    tst.l     d0
    beq.s     MSEnd
    movea.l   d0,a1
    move.l    #ioa_SIZEOF,d0
    jsr       _LVOFreeMem(a6)
    move.w    #$f00,$dff180
    bra.s     MSLp1

MSEnd:
    move.w    #1,_listEmpty                  ; Liste wieder leer!
    jsr       _LVOEnable(a6)
    movem.l   (a7)+,d0-a6
    move.l    _OldSwitch,-(a7)               ; in Switch() einspringen
    rts

#endasm
```

Taken from Amiga-Manuals-Website

Von der Idee zum Sound 4

Die bisherige Lektüre dieses Buches konnte wahrscheinlich davon überzeugen, daß der Amiga mit seinen vier Sampler-Stimmen über eine ungewöhnlich leistungsfähige Klangerzeugung verfügt. Ungewöhnlich sind dementsprechend auch die Techniken und Methoden, mit denen man dem Amiga sein musikalisches Potential entlocken muß. Angefangen von der Gewinnung geeigneter Sound-Daten über die Instrument-Erstellung bis hin zur Komposition, alle jeweils notwendigen Schritte haben mit der Arbeit an normalen Synthesizern kaum etwas gemein und verlangen vom Anwender einige Erfahrung. Zwar ist es mit Hilfe der angebotenen Software ohne weiteres möglich, eine gesamplete Soundpassage oder ein kleines Liedchen aus dem Lautsprecher erklingen zu lassen; die erzielten Ergebnisse lassen am Anfang jedoch meist zu wünschen übrig. Um an die beeindruckende Qualität z.B. der Titelmelodien vieler neuer Videospiele und Intros heranzukommen, muß man sich hinsichtlich der Wahl der musikalischen Elemente und der Kompositionstechnik weitgehend den hier herrschenden Verhältnissen anpassen.

Da sich durch die mit diesem Buch gelieferte Software auch für den Programmierer ein Weg zur freien Verwendung seiner Kompositionen erschließt, soll in diesem Kapitel hauptsächlich auf die Arbeit mit den üblichen Programmen wie z.B. Sonix eingegangen werden. Im Mittelpunkt sollen hierbei hauptsächlich die Tricks und Methoden stehen, die zur Überwindung eventueller Unzulänglichkeiten der Software notwendig sind, von den Handbüchern aber verschwiegen werden.

4.1 Sampling

Da die mitgelieferten Sounds der gängigen Kompositionsprogramme meist ziemlich dünn klingen und daher nur beschränkt verwendbar sind, steht am Beginn der Arbeit des Musikschaffenden wohl die Beschaffung passender Instrumente. Zwar bieten z.B. viele Public-Domain-Sammlungen eine große Auswahl hiervon, das Salz in der Suppe sind jedoch nur die speziell für den jeweiligen Zweck entworfenen Klänge. Das Programm Sonix ragt in dieser Beziehung besonders heraus; hier ist sowohl ein ganz passabler Analog-Synthesizer als auch eine Möglichkeit zur Beeinflussung von Sampled Sounds vorhanden. Fehlen solche Features, wie z.B. bei D.M.C.S., so rückt die Instrument-Erstellung via Sampling noch weiter in den Vordergrund.

Dies zeigt bereits, daß für die Erzielung akzeptabler Ergebnisse teilweise einige Umwege in Kauf genommen werden müssen, wie z.B. die Verwendung mehrerer Sampler-Programme. Aus diesem Grund sind die Anweisungen und Ratschläge im vorliegenden Abschnitt auch so allgemein wie möglich gehalten. Hier soll zunächst die Entstehung eines Samples, von der Signalquelle bis zur Aufnahme, eingehend erläutert werden.

4.1.1 Die Signalquelle

Mikrofon

Um »natürliche« Klangquellen wie z.B. akustische Instrumente oder Stimmen aufnehmen zu können, benötigt man in jedem Fall ein Mikrofon, das den Schalldruck in elektrische Spannung umwandelt. Benötigt man es ausschließlich für den Sampling-Einsatz, so kommen hier bereits recht preiswerte Modelle um die 100–200 Mark in Frage. Diese und auch noch im Preis darüberliegende Geräte sind meistens als Kondensator-Mikro mit Nierencharakteristik ausgeführt, d.h., die Empfindlichkeit des Mikros ist in allen Richtungen ungefähr gleich. Das führt uns auch gleich zu der Hauptanforderung an den Aufnahme-Raum: Dieser muß akustisch möglichst trocken sein, also wenig Nachhall aufweisen. Ansonsten empfängt ein Mikrofon mit breiter Charakteristik zuviel Reflexionsgeräusche, die sich meist sehr störend auswirken. (Im Köllner Dom mags ja ganz interessant klingen.) Mit einem guten Gehör läßt sich der Nachhall eines Raumes sehr einfach prüfen, indem man ein paarmal kräftig in die Hände klatscht. Ein möblierter Raum mit schweren Vorhängen ist einem leeren auf jeden Fall vorzuziehen; eine billige, aber recht effektive Maßnahme ist auch das Auskleiden eines Raumes mit weichem Dämm-Material wie z.B. Steinwolle oder Eierkartons.

Eine zweite Eigenschaft speziell der Kondensator-Mikrofone ist es, nur eine recht geringe Ausgangsspannung abzugeben. Möchte man ein Mikro direkt an den Digitizer anschließen, so sollte man beim Kauf desselben beachten, daß dieser auch über einen genügend großen Eingangspegel-Bereich verfügt. Hier werden hohe Ansprüche an die im Digitizer verwendeten Bauteile gestellt, da sie auch bei niedrigem Eingangspegel und der dementsprechend hohen Verstärkung noch ein passables Rauschverhältnis ergeben sollten.

Elektrische Instrumente

Auch elektrische (E-Gitarre, E-Baß) und elektronische Instrumente (Synthesizer) lassen sich im allgemeinen problemlos an einen Digitizer anschließen. Deren Ausgangsspannung liegt in der Regel etwas höher als die eines Mikrofons, ist aber trotz allem noch relativ niedrig; bei Digitizern ganz ohne Eingangsregler ist auch hier ein Vorverstärker erforderlich. Bei einer E-Gitarre, die einen sehr hohen Dynamikbereich besitzt, kann die Verwendung eines Dynamik-Kompressors zur einfacheren Aussteuerung sehr hilfreich sein. Das erleichtert auch dem Gitarristen die Einhaltung des gewünschten Lautstärkepegels. Bei Synthesizer-Keyboards sollte man, wenn möglich, den Einfluß von Velocity und Aftertouch ganz abschalten, damit man eine einheitliche Lautstärke erhält. Um dem Sound mehr Fülle zu geben, empfiehlt sich manchmal die Verwendung eines leichten Hall-Effekts.

Touch-Pickups

Für viele akustische Instrumente wie Gitarre, Geige oder auch Schlagzeug gibt es speziell konstruierte (nichtmagnetische) Tonabnehmer, die ähnlich einem Mikrofon die mechanischen Schwingungen des Klangkörpers in ein elektrisches Signal umwandeln, aber direkt am Instrument montiert werden. So stellt z.B. die Firma Ovation seit langem akustische Gitarren mit eingebauten Tonabnehmer her, der ein dem natürlichen Klang sehr ähnliches Signal liefert. Exakt den gleichen

Höreindruck wie eine Mikrofonaufnahme liefern sie jedoch nicht; in professionellen Studios werden deshalb die Signale aus diesen Tonabnehmern mit denen aus dem Mikrofon gemischt, um einen möglichst lebendigen Klang zu erhalten.

Tonband und CD-Player

Selbstverständlich können auch die meisten HiFi-Komponenten wie Tape-Decks und CD-Player direkt an einen Digitizer angeschlossen werden, sofern dieser vom Eingangsbereich her dafür ausgelegt ist; die Norm-Spannung liegt hier ungefähr bei 1 Vss, also etwas höher als bei den Musikinstrumenten. Plattenspieler scheiden hier aus, da sie meist einen Entzerrer benötigen.

Stereoverstärker

Die wohl beste und flexibelste Möglichkeit für den Anschluß eines Digitizers ist der (Vor-) Verstärker einer Stereoanlage. Hier liegen die Eingangs-Signale in mehreren Formen an: Als Norm-Signal an den Monitor Out- bzw. Tape Out-Anschlüssen sowie als verstärktes Signal an den Lautsprecherausgängen. Letztere sind jedoch stets mit Vorsicht zu genießen, schon so mancher Digitizer hat hier sein junges Leben vorzeitig ausgehaucht.

Passender ist da schon der fast überall vorhandene Kopfhörer-Ausgang, an dem lange nicht so hohe Spannungen anliegen, der aber auch über die Lautstärke- und Klangregelung beeinflußt werden kann. Diese können für die Aussteuerung bzw. für die Verminderung von Aliasing-Frequenzen verwendet werden. Das ist zwar mit ein wenig Bastelei verbunden (es sei denn man besorgt sich einen Klinke-zu-Cinch-Adapter), lohnt aber den Aufwand; und wenn man nicht gerade einen echten Stereo-Sampler verwenden will, können die beiden Stereo-Kanäle zum direkten Abhören sowohl des originalen als auch des gesampleten Signals eingesetzt werden. Dies sind die dafür nötigen Schritte:

1. Um aus den zwei Stereo-Signalen eines HiFi-Gerätes möglichst ein Mono-Signal zu machen, kann man die beiden Leitungen meist einfach zusammenschalten, ohne daß dies für das Gerät gefährlich würde; wer ganz sicher gehen will, sollte nochmal beim Händler nachfragen. Auf keinen Fall darf dies jedoch mit Verstärker-Ausgängen, wie den Kopfhörer- oder Lautsprecherausgängen gemacht werden! Diese könnten dabei Schaden nehmen!

2. Das so gewonnene Mono-Signal gebe man nun auf einen der beiden Stereo-Kanäle und aktiviere den gewählten Eingang am Verstärker.

3. Man verbinde den entsprechenden Kanal am Kopfhörer-Ausgang mit dem Digitizer und regle dessen Eingangsempfindlichkeit auf das Minimum, also auf die höchste Eingangsspannung. Zur Sicherheit sollte auch der Lautstärkeregler am Verstärker vorher auf 0 gestellt werden.

4. Jetzt muß derjenige Audio-Ausgang des Amiga, der das zum Abhören gedachte Signal führt, mit dem anderen Stereo-Eingang des Verstärkers verbunden werden. Wenn man die Monitor-Funktion seines Sampler-Programmes aufruft, sind sowohl das Original sowie das digitalisierte Signal zu hören und die Aussteuerung kann vorgenommen werden. Mit ein wenig Herumprobieren am Lautstärke- und Balanceregler kann man jetzt leicht die optimalen Verhältnisse für Aufnahme und Mithör-Kontrolle einstellen.

4.1.2 Störeinflüsse und deren Beseitigung

Um ein akzeptabel klingendes Sample zu erhalten, muß natürlich auch das am Digitizer eintreffende Eingangssignal möglichst frei von allen Störgeräuschen sein. Bei der Aufnahme mit einem Mikrofon sind da natürlich zuerst die manchmal schwer vermeidbaren Nebengeräusche zu nennen, wie z.B. Straßenlärm, irgendwo klingelnde Telefone, Stuhlquietschen oder derlei unerfreuliche Dinge mehr. Auch der höchst nervige Ventilator eines Amiga 2000 macht eine saubere Mikrofonaufnahme meist unmöglich; um dies zu vermeiden, empfiehlt sich eine vorherige Aufnahme des gewünschten Klanges auf ein Tonband. Die Lautstärke am Verstärker (oder am Monitor) muß dabei auf 0 gestellt werden, um Rückkopplungen zu verhindern. Störender Nachhall kann, wie oben beschrieben, nur durch die akustische Trockenlegung des Aufnahmeraumes vermieden werden. Allgemein gilt (nach Murphy), daß im Sample letztendlich jedes noch so kleine Störgeräusch wahrgenommen wird, auch wenn es bei der Aufnahme noch gar nicht auffiel.

Noch wesentlich tückischer sind allerdings die elektrischen Störsignale, wie z.B. das berüchtigte Netzbrummen. Es entsteht durch die 50-Hz-Wechselspannung im Stromnetz, die bei ungünstigem Aufbau oder Verschaltung von Audio-Geräten in den Signalweg einstreuen können. Um solche Ärgernisse zu umgehen, sollte man von vorn herein die folgenden Regeln beachten:

❑ Für sämtliche Verbindungen sollten nur abgeschirmte Hifi-Kabel verwendet werden; ein Lautsprecherkabel z.B. ist ungeeignet.

❑ Achten Sie darauf, daß sämtliche Anschlüsse sauber und frei von Korrosion sind. Gegebenenfalls müssen diese mit Watte und Alkohol bzw. mit Schleifpapier gereinigt werden.

❑ Prüfen sie, ob alle Leitungen und Lötstellen intakt sind. Eine lose Massenverbindung kann ein sehr lautes Brummen verursachen, eine gebrochene Signalleitung läßt einfach nichts mehr durch. Sollten sie ein bewegliches Instrument wie z.B. eine Gitarre verwenden, sollten Sie alle Kabel vor dem Kauf auf Knister-Freiheit testen, indem Sie sie an einen Verstärker anschließen und ein wenig herumbewegen.

❑ Die Geräte und Verbindungsleitungen sollten sich nicht in der Nähe von starken Brummquellen wie Trafos, Elektromotoren, Dimmern oder auch Neonröhren befinden. Teilweise kann es auch nötig werden, den Monitor auszuschalten.

❑ Der Digitizer sollte stets auf die geringste Eingangsempfindlichkeit gestellt sein, damit das Verhältnis zwischen Nutz- und Störsignalen so günstig wie möglich ausfällt.

Sollte trotz der korrekten Einhaltung aller dieser Regeln immer noch ein unerklärliches Brummen oder ähnliches vorhanden sein, so hilft nur die Methode der sog. Signalwegverfolgung: Jedes Glied in der Komponentenkette, angefangen vom Digitzer bis zurück zur Signalquelle, muß auf sein Ausgangssignal hin geprüft werden. Zuerst testet man den Digitizer ohne Beschaltung, dann mit dem ersten Zwischenglied, dann mit dem nächsten verbunden usw. Wenn der Digitizer allein schon kein sauberes 0-Signal hinbekommt, so wurde er wahrscheinlich beim Transport beschädigt und sollte umgetauscht werden.

4.1.3 Die Aufnahme

Stimmung und Sampling-Rate

Vor allem für den Einsatz in Kompositionsprogrammen muß ein Sample sehr genau auf die dort verwendeten Wiedergaberaten abgestimmt sein. Betrachten wir zunächst den Fall eines tonalen Instruments, mit dem Melodien gespielt werden sollen. Die meisten Programme wie Sonix und D.M.C.S. spielen das C mit einer Rate von 8363 Hz, was hier die niedrigste verwendete Rate ist. Im allgemeinen wird nur eine Oktave verwendet; der in diesem Fall höchste Ton, das H, wird mit einer Rate von 15838 Hz wiedergegeben. Dies ist leider die höchste Sampling-Rate, die die gängigen Kompositionsprogramme zulassen. Ein Instrument, welches zusammen mit anderen gespielt werden soll, sollte sich an diese Vereinbarungen halten, damit es die eingegebenen Noten korrekt wiedergibt. Ein bereits existierendes, nicht korrekt gestimmtes Sample muß dazu erst in die richtige Tonlage gebracht werden; hierzu besitzen viele Sample-Edit-Programme spezielle Funktionen, wie die Resample-Funktion des AudioMaster. Diese verursachen aber durch den Umrechnungs-Vorgang meist eine inakzeptable Verschlechterung der Klangqualität, die nur durch eine vorherige Filterung vermieden werden kann.

Wesentlich besser ist es hingegen, wenn das Sample gleich korrekt gestimmt und bei der richtigen Sampling-Rate aufgenommen wird. Ersteres ist mit der Tuning-Tone-Funktion des Audiomaster sehr einfach, die einen Ton von der Höhe der gerade eingestellten Target-Rate erklingen läßt. Nach diesem Ton kann das verwendete Musikinstrument gestimmt werden, ansonsten muß man vor der Aufnahme ein möglichst reines, stimmgenaues Sample laden und sich an diesem orientieren. Ein Band oder CD-Player kann jedoch in der Regel nicht in seiner Tonhöhe variiert werden; in diesem Fall ist die Sampling-Rate entsprechend zu korrigieren. Hier hilft der Tuning Tone des AudioMaster enorm: Dieser braucht nur auf den Klang des Tonbands gestimmt zu werden, und schon kann man die korrekte Sampling-Rate im Fenster ablesen.

Zur Wahl der Sampling-Rate selbst sind hier jedoch noch einige Bemerkungen zu machen. Wie gesagt, verwenden die Kompositionsprogramme ein Sample für eine ganze Oktave. Wurde ein Ton sehr tief (z.B. C) oder sehr hoch (z.B. H) aufgenommen, so wird er sich am jeweils anderen Ende der Oktave stark verfremdet anhören, womit sich auch eine MickyMaus- bzw. eine Rübezahlstimme erzeugen lassen. Dies ist jedoch oft nicht erwünscht, da ein Instrument ja meist in allen Tonlagen so natürlich wie möglich klingen soll. Deshalb wählt man am besten einen mittleren Ton, z.B. das F oder das G. So erreicht man, daß das Sample auf den meisten Tönen der Oktave ganz passabel klingt.

Im Gegensatz zu tonalen Instrumenten möchte man manchmal längere Samples als sog. Backup-Tracks oder auch kurze perkussive Sounds einsetzen, die alle nur auf einer bestimmten Rate wiedergegeben werden müssen. Hierfür sollte man am Besten die höchstmögliche Rate (15838 Hz) wählen, um eine optimale Klangqualität zu erzielen. Auch bei solchen Zwecken muß die Tonquelle vorher an die Sampling-Rate angepaßt werden, oder gegebenenfalls umgekehrt.

In der Praxis gestaltet sich der hier vorgeschlagene Weg jedoch etwas umständlich: Leider besitzt der AudioMaster in der vorliegenden Version keine Möglichkeit zur Änderung der Aufnahme-Rate (was hoffentlich in der Version 3.0 behoben sein wird). Aus diesem Grund wurde die

Aufnahme der oben genannten Instrumente mit dem Programm De-Luxes-Sound, und erst die weitere Bearbeitung mit dem AudioMaster durchgeführt. Im Extremfall, z.B. wenn erst die richtige Sampling-Rate für einen Klang auf Band ermittelt werden soll, muß man zuerst in den AudioMaster, hier mit Hilfe des Tuning Tones die Rate bestimmen, dann in ein anderes Programm und die Aufnahme durchführen, und schließlich wieder im AudioMaster den Feinschliff vornehmen.

Aussteuerung und Nullabgleich

Um ein optimales Signal-/Rauschverhältnis zu erzielen, muß das Eingangssignal so hoch wie möglich ausgesteuert werden. Um die Grenze des Wertebereiches und damit der Verzerrung genau bestimmen zu können, verfügen die meisten Sampler-Programme über eine spezielle Einrichtung, die zumindest eine evtl. Bereichsbeschreitung anzeigt. Der Audiomaster besitzt hierfür sein komfortables Oszilloskop, das man durch einen Klick auf den rechten Mausknopf in Ruhe betrachten kann. Man wird feststellen, daß sehr kleine »Berührungen« des unteren und oberen Randes noch nicht sonderlich ins Gewicht fallen; jedoch spätestens wenn ein mehrere Werte langes Clipping auftritt, wird sich das als deutliche Verzerrung bemerkbar machen.

Für die optimale Ausnutzung des Wertebereichs ist es außerdem wichtig, daß der Digitizer bei einer Spannung von 0 Volt auch tatsächlich 0-Werte liefert. Liegt die Ruheposition unter oder über dem Wert 0, so stößt die Spannung eher an eine der Bereichsgrenzen und kann begreiflicherweise dann nicht mehr so hoch ausgesteuert werden. Die meisten Digitizer sind ab Werk korrekt eingestellt, manche aber besitzen hierfür ein Trimm-Poti, welches vor dem Sampeln justiert werden muß.

Der Start des Sampling-Vorganges

Um den zur Verfügung stehenden Speicherplatz voll nutzen zu können, muß die Aufnahme möglichst genau übereinstimmend mit dem Beginn des Klanges gestartet werden. In diesem Punkt offenbart sich eine weitere Schwäche des AudioMaster: Hier kann nur mit Hilfe der Maus gestartet werden, was oft einen zu frühen oder zu späten Beginn des Samples zur Folge hat. Einige Programme, wie auch De-Luxes-Sound, besitzen hierfür eine Funktion, die die Aufnahme automatisch auslöst, sobald der Eingangspegel einen bestimmten Wert überschreitet. Dies ist oft sehr praktisch, da hiermit kein Byte Speicher verschenkt und der Klang trotzdem immer vollständig aufgenommen wird. Man sollte aber stets den geringstmöglichen Auslösewert verwenden, da sonst unter Umständen doch Teile des Anschlags der Wellenform verlorengehen könnten. Da das Sample mit dieser Methode niemals wirklich mit einer 0 beginnt, kann es nötig werden, den Anfang des Samples bis zum nächsten Nulldurchgang wegzuschneiden.

4.2 Die Erstellung von Instrumenten

Ist das gewünschte Sample nun glücklich im »Kasten«, kann man die Nachbearbeitung in Angriff nehmen. Für diesen Zweck ist wohl nach wie vor der Audiomaster zu empfehlen, da hier eine große Anzahl von Funktionen zur Bearbeitung und Korrektur von Sample-Daten vorhanden sind, mit denen dem Klang mit ein wenig Geschick viel Lebendigkeit und Fülle verliehen werden können.

4.2.1 Korrektur der Lautstärke

Hüllkurven

Wenn man ein Sample zu einem Instrument verarbeitet, so stellt sich stets die Frage, ob dieses nur mit der gegebenen Hüllkurve abgespielt, oder mit einer zusätzlichen Hüllkurve versehen werden soll. Sowohl im Sonix- als auch im IFF-Format ist dies zumindest theoretisch vorgesehen. Wenn aber ein Instrument mit einer Hüllkurve beaufschlagt werden soll, so ist es ratsam, aus den Sample-Daten vorher möglichst jede eigene Hüllkurve herauszubügeln, da es sonst zu schwer vorhersehbaren Ergebnissen kommt. Im Klartext bedeutet das, daß das Sample eine vom Beginn bis zum Ende möglichst konstante Lautstärke aufweisen muß.

Hier ist die Change-Volume-Funktion des Audiomaster überaus hilfreich, da sie die kontinuierliche Veränderung des Lautstärkepegels in einem gewählten Bereich erlaubt. Möchte man z.B. einen langsamen Anschlag in dieser Weise korrigieren, so braucht man lediglich den Bereich vom Beginn des Samples bis zum Erreichen der maximalen Lautstärke anzuwählen und dann die Start-Lautstärke auf das Maximum (200) und die Endlautstärke auf gleichbleibend (100) zu stellen. Gegebenenfalls muß dies mehrfach durchgeführt werden, da eine bloße Verdoppelung am Beginn oft nicht ausreicht.

Konstante Haltephasen

Ein Spezialfall in der Anwendung der Change-Volume-Funktion ist die Herstellung eines konstanten Lautstärkepegels innerhalb des Repeat-Teils geloopter Instrumente. Digitalisiert man beispielsweise ein Klavier oder eine Gitarre, so nimmt die Lautstärke nach dem Anschlag ja ständig ab, was dazu führt, daß nahezu an allen Punkten des Samples ein hörbarer Unterschied zwischen dem Beginn und dem Ende des Repeat-Parts besteht. Gerade hier ist das Gehör jedoch leider besonders heikel, nur allzuleicht erkennt es die Übergänge und empfindet einen störenden Monotonitäts-Effekt (boing-boing-boing-boing). Am besten geht man hier so vor: Man wählt als Beginn der Repeat-Parts eine Stelle im Sample, die eine noch nicht allzu geringe Lautstärke aufweist, um einen möglichst großen Spielraum für evtl. Hüllkurven-Manipulationen zuzulassen. Ab hier hebe man den Pegel mit Hilfe der Change-Volume-Funktion so an, daß er über eine möglichst lange Phase annähernd konstant ist, damit auf einer möglichst großen Länge nach einem passenden Loop-Point gesucht werden kann. Dabei muß der Endwert sehr gefühlvoll eingestellt werden, der Startwert sollte 100% betragen. In jedem Fall werden wohl einige Versuche nötig sein, bis der Loop überzeugend klingt.

4.2.2 Korrektur der Tonhöhe

Bereits bestehende Samples müssen zuweilen einer Tonhöhenkorrektur unterzogen werden, wenn sie nicht mit den üblichen Wiedergaberaten übereinstimmen. Hierfür ist z.B. die Resample-Funktion des Audiomaster sehr nützlich, die das Sample auf eine neue Rate umrechnet. Vor allem bei der Umrechnung auf niedrigere Raten ist jedoch zu beachten, daß auch hier wie bei der Aufnahme weitere Aliasing-Komponenten erzeugt werden, abhängig vom Frequenzgehalt des Originals und der Ziel-Rate. Es gelten hier also die gleichen Regeln wie bei der Digitalisierung selbst, nämlich daß das Frequenzspektrum des Originals begrenzt werden muß, um bei einer

bestimmten Ziel-Rate keine Aliasing-Frequenzen in den Nutzbereich, der ja durch den Ausgabe-Filter bestimmt wird, fallen zu lassen. Leider existiert für den Amiga bisher kein Sample-Edit-Programm, welches einen wirklich brauchbaren digitalen Filter besitzt. Dies kann sich allerdings schnell ändern, da die Realisation eines solchen im allgemeinen kein Problem darstellt; die theoretischen Grundlagen dafür sind schon in den 60er Jahren hinreichend ausgelotet worden und stehen in zahlreichen Werken der Öffentlichkeit zur Verfügung.

4.2.3 One-Shot- und geloopte Instrumente

Viele Sounds, wie z.B. Percussion- und Drumsounds oder andere hinreichend kurze Klänge, benötigen keine konstante Haltephase und werden nur einmal wiedergegeben. Ihre Herstellung gestaltet sich daher sehr einfach: Lediglich Korrekturen der Sampling-Rate und/oder Amplitude müssen zuweilen vorgenommen werden, um ein einsatzbereites Instrument zu erhalten. Andere Klänge wiederum wie Orgel oder Flöte besitzen jedoch eine solche Sustain-Phase, in der der Ton meist einen relativ konstanten Pegel aufweist. Dies ist mit der Audio-Einheit des Amiga nur zu bewerkstelligen, indem ein bestimmter Datenblock in ständiger Wiederholung ausgegeben wird, vergleichbar mit einem Stück Tonband, dessen Enden zu einer Endlosschleife (Loop) zusammengeklebt werden. Im IFF- und Sonix-Format besteht ein Instrument aus einem One-Shot-Teil, der beim Beginn einer Note einmal gespielt wird, und eben einem solchen Repeat-Teil, der anschließend so lange wiederholt wird, wie die Note andauert.

Looping und natürliche Periodizität

Die Bestimmung eines solchen Datenblocks ist bei natürlichen Klängen ziemlich kompliziert, da sie nur äußerst selten genau periodisch verlaufen. Da wir dem Klang jedoch eine Periode aufzwingen müssen, sollten wir dazu entweder eine vollkommen gleichklingende Soundpassage oder eine Stelle mit einer natürlichen Periode wählen. Ersteres ist wohl am schwierigsten zu bewerkstelligen: Wie schon im letzten Abschnitt angedeutet, müßten sowohl die Lautstärke als auch die spektrale Verteilung genau konstant sein, um einen wirklich »stehenden« Klang zu erreichen. In der Praxis ist das aber auch gar nicht so wünschenswert, da sich Lebendigkeit und Tiefe eines Instruments nur mit Hilfe einer gewissen Klangvariation erreichen läßt, sei es nun ein Tremolo, Vibrato, Chorus oder was auch immer. Die Kunst der Erstellung geloopter Instrumente besteht also hauptsächlich im Auffinden eines Datenblocks, dessen Beginn und Ende sich klanglich möglichst ähnlich sind.

Um dies zu veranschaulichen, betrachten wir zunächst einen relativ einfachen Fall, z.B. eine professionell gespielte Flöte. Der Ton setzt mit einem typischen Anblasgeräusch ein, bleibt eine kurze Zeit konstant, um dann in ein leichtes Vibrato überzugehen. Hier wird man den Teil bis zum vollen Einsetzen des Vibratos als One-Shot-Part wählen; der optimale Repeat-Part muß nun innerhalb der anschließenden Daten ermittelt werden, wobei man natürlich die Periode des Vibratos als Zeitbasis für den Loop verwenden sollte. Das heißt, daß der Repeat-Datenblock genau die Länge von einer oder mehreren Vibrato-Schwingungen haben muß, damit die Tonhöhe am Beginn und am Ende des Blocks möglichst gleich ist.

Schwieriger wird es, wenn der Ton beim besten Willen keine erkennbare Periode aufweist; in diesem Fall wird die Erstellung eines Loops erst wirklich zum kreativen Akt, da hier oft viele

Experimente mit verschiedenen Loop-Positionen, Pegelkorrekturen, Filteralgorithmen (wenn vorhanden) und ähnlichem vorgenommen werden müssen, bis man überzeugende Ergebnisse erzielt. Da ein Loop ja in den meisten Fällen den Eindruck eines durchgehend präsenten, aber doch möglichst lebendigen und bewegten Klanges erzeugen soll, muß man seine Aufmerksamkeit hier auf jede noch so kleine Einzelheit im spektralen Verlauf richten. Zum Beispiel kann schon eine kleine Schwebung im Ton eines Klaviers (wo ja bis zu drei Saiten angeschlagen werden) die Basis für einen guten Loop bieten, solange die Lautstärke- und Klangfarbenunterschiede zwischen Beginn und Ende entsprechend gering ausfallen oder kompensiert werden können.

Übergänge und Phasensprung

Nachdem eine zumindest theorethisch hoffnungsvolle Loop-Position gefunden ist, wird in der Regel noch einige Feinarbeit nötig sein. Auch am Übergang zwischen Beginn und Ende eines Datenblocks werden die einzelnen Werte ja einfach hintereinander ausgegeben, so daß der Loop auch auf der Ebene der einzelnen Sample-Werte harmonisiert werden muß. Besteht zwischen dem Anfangs- und dem Endpunkt des Loops ein zu großer Wertunterschied, so wird sich das als ein lästiges Click-Geräusch bemerkbar machen. Eine wichtige Anforderung an einen Loop-Block ist also, daß die Sample-Werte an den Blockgrenzen möglichst gleich sein müssen. Der Audiomaster z.B. bietet zur Auffindung solcher Stellen seine Seek-Zero-Funktion an, mit der man eine Block-Markierung auf die nächste (erreichbare) Nullstelle setzen kann. In der Praxis ist es jedoch nicht unbedingt notwendig, daß der Übergang auf einer 0 erfolgt; für einen Klick-freien Loop ist es am wichtigsten, daß Beginn und Ende die gleichen Werte aufweisen und fließend ineinander übergehen.

Hierbei kommt es jedoch nicht nur auf die beiden Sample-Werte direkt an den Grenzpunkten an, sondern auf den ganzen Spannungsverlauf in der Umgebung des Übergangs. Stellen wir uns nur einmal eine einfache Sinuswelle vor, die geloopt werden soll, wobei wir den Beginn des Loops einmal willkürlich auf den 0-Punkt setzen wollen, ab dem die Kurve zuerst stetig ansteigt. Den Endpunkt sollte man nun natürlich ebenfalls auf einen 0-Wert setzen. Hier stehen jedoch zwei Nullpunkte mit unterschiedlichen Eigenschaften zur Auswahl: ein Punkt, ab dem die Werte ansteigen, und einer, ab dem sie absinken. Die Entscheidung liegt hier auf der Hand, selbstverständlich wird man einen Punkt wählen, in dem auch die Steigung der Kurve (mathematisch: die erste Ableitung) der am Beginn des Loops möglichst gleich ist. Wird diese Regel vernachlässigt, so kommt es zu einem Phasensprung, der im Moment der Wiedergabe zu einer veränderten Spektralverteilung und somit meist ebenfalls zu einem hörbaren Übergangs-Klick, -Plop oder -Bumm führt. Bei natürlichen Samples gibt es aber wie gesagt oft keinerlei Periodizität, die Wellenformen an den gewählten Beginn- und Endpunkten weisen meist nicht die mindeste Ähnlichkeit zueinander auf. Gerade dann wird die Auswahl der Loop-Positionen besonders heikel: Liegt der Anfangspunkt z.B. auf einem langen, steilen Ramp nach oben, so tut man gut daran, den Endpunkt ebenfalls auf eine vergleichbare Position zu legen. Die Maxime lautet immer, einen Loop so unauffälig wie irgend möglich zu gestalten.

Längen-Raster und Korrekturen

Die Erstellung von guten Loop-Instrumenten wäre in den meisten Fällen ein Kinderspiel, wäre man nicht hinsichtlich der Byte-Längen von One-Shot- und Repeat-Part an bestimmte format-

bedingte Raster gebunden. Da der Amiga immer nur zwei Sample-Werte auf einmal verarbeiten kann, müssen die Part-Längen gerade sein, so daß schon einmal jedes zweite Byte als Loop-Position ausfällt. Schlimmer wird es, wenn man z.B. im Audiomaster eine mehroktavige Wellenform zugrunde legt, so daß nur noch jedes vierte bzw. jedes achte Sample in Frage kommt, da die höheren Oktaven ja auch eine geradzahlige Länge haben müssen. Bei einem Sonix-Instrument werden die Verhältnisse vollends katastrophal, hier kann wegen des Speicherformats nur jeder 32. Wert als Loop-Point verwendet werden. (Das ist nicht sehr schlau, lieber Mark Riley, und dazu noch vollkommen unnötig!) Die Seek-Zero-Funktion des Audiomaster ist in diesem Fall unbrauchbar; es beginnt eine lange und zermürbende Suche nach Punkten innerhalb dieses Rasters, die die oben genannten Bedingungen erfüllen. Hier einige Tips für die Suche:

❑ Natürliche Samples sind in den seltensten Fällen wirklich ganz exakt gestimmt, so daß die Periode einer solchen Wellenform sich in deren Verlauf oft mehrmals mit dem Datenraster in der Phase überschneidet. Sucht man an einer Wellenform entlang, so wird man fast immer Stellungen finden, in der beide Loop-Points an vergleichbare Positionen fallen.

❑ Das Längen-Raster selbst läßt sich zwar nicht verändern; es kann jedoch die Stellung verändert werden, in der die Klangdaten auf dem Raster zu liegen kommen. Dazu können entweder Daten am Beginn des Samples herausgeschnitten oder auch 0-Daten hinzugefügt werden, was meistens keinen hörbaren Effekt hat.

❑ Glaubt man einen vom gesamten Klangverlauf her idealen Loop-Datenblock gefunden zu haben, kann in dieser Umgebung beim besten Willen keine geeigneten Loop-Points finden, so hilft nur manuelle Nachbearbeitung des Samples. Hierzu bietet sich z.B. die Edit-Freehand-Funktion des AudioMaster an. Paßt z.B. der Aufwärts-Ramp an dem durch das Raster vorgegebenen Endpunkt nicht ganz genau in den ausgewählten Beginn des Loops, so kann die lästige Differenz mit ein wenig Geschick von Hand begradigt werden, so daß weder Klick-trächtige Werte- noch Phasensprünge entstehen.

4.3 Komposition

Zur Verwirklichung unserer musikalischen Vorstellungen steht uns der wichtigste Teil der Arbeit aber noch bevor: Die Verteilung der Noten auf die einzelnen Stimmen, Arrangement und Instrumentierung des Stückes bilden wohl die schwierigste Komponente und setzen auch am meisten Kreativität und musikalisches Können voraus. Erschwerend wirkt sich hier natürlich die geringe Anzahl der Stimmen aus, die das Erzielen eines vollen, satten Klangbildes oft zu einer Puzzle-Arbeit werden läßt. Die in diesem Abschnitt gegebenen Informationen sollen vor allem der Überwindung dieser Einschränkung sowie der eventuellen Unzulänglichkeiten der Software dienen.

4.3.1 Sonix-Tips und -Tricks

Gebundene Noten

Leider besitzt Sonix die unangenehme Eigenschaft, sämtliche Noten um ein Viertel kürzer zu spielen als angegeben. Dies fällt meistens nicht so sehr auf, da die Hüllkurve der meisten

Instrumente eine gewisse Ausschwingzeit vorsieht. Manchmal jedoch, wie z.B. für die Nachahmung mehrerer Gitarrenanschläge, ist eine möglichst kontinuierliche Wiedergabe gewünscht. Darüber hinaus läßt Sonix keine längeren Notenwerte als eine ganze Note zu.

Diesen beiden Übeln kann hier nur mit einer Veränderung der Hüllkurve abgeholfen werden, welche allerdings nur bei den Analog- und bei den Sampled-Sounds zur Verfügung steht; IFF-Instrumente müssen stets zuerst umgewandelt werden. Grundsätzlich hat man die Wahl zwischen zwei Methoden: Entweder kann die Ausschwingzeit verlängert werden, indem man die Release-Rate (Rate 4) auf einen sehr niedrigen Wert setzt, oder man setzt das Release-Level auf einen Wert über 0, so daß der Ton beliebig lange nachschwingen kann. Möchte man also z.B. ein Instrument über drei ganze Noten binden, so braucht man sich lediglich eine Kopie davon zu erstellen, die man in der oben genannten Weise modifiziert. Dieses neue Instrument muß vor der betreffenden Note aktiviert werden; dahinter läßt man so viele Pausen folgen, wie dem gewünschten Notenwert entspricht. Um eine solche Note wieder zu stoppen, muß entweder eine neue gespielt werden (vorzugsweise wieder mit dem Original-Instrument), oder ein Instrument mit einer Lautstärkeeinstellung von 0, wenn eine echte Pause gewünscht ist.

Dynamik

Da Sonix in der Noteneingabe keine Möglichkeiten zur Dynamikvariation vorsieht, muß auch hierfür der Lautstärke-Parameter des Instruments herhalten. Dazu müssen wiederum erst mehrere Duplikate des Instruments angefertigt werden, die mit verschiedenen Volume-Einstellungen versehen und dann bei Bedarf aktiviert werden. Dem um seinen wertvollen Speicherplatz besorgten Anwender (oder Programmierer) werden hier sicher die Haare zu Berge stehen, aber halb so wild: Für jedes Analoginstrument gehen lediglich 506, für eine Version eines Sampled Sound gar nur 132 Byte verloren. Bei ersteren ist allerdings darauf zu achten, daß die DCO-Wellenform nicht verändert wird, da für jede neue Wellenform ein eigenes Filterarray mit jeweils 64 verschiedenen gefilterten Versionen erstellt wird; dies nimmt natürlich eine Menge Speicher in Anspruch, immerhin ganze 16 Kbyte.

Nachhall

Vor allem Instrumente mit einer langen Ausschwingzeit können erheblich an Authentizität verlieren, wenn sie in zu schneller Folge angespielt werden, da jede Note ja beim Beginn einer weiteren schonungslos unterbrochen wird. Dem kann man nur dadurch entgehen, indem man die aufeinanderfolgenden Noten der Reihe nach auf verschiedene Stimmen legt, soweit dies deren momentane Auslastung zuläßt. Versucht man z.B. den Klang einer Gitarre oder eines Klaviers mit gedrücktem Pedal nachzuahmen, so wird man in der Regel sicher drei Stimmen für diesen Effekt opfern müssen, damit das Abschneiden einer Note zugunsten einer neuen so wenig wie möglich auffällt.

4.3.2 Schlagzeug-Tracks

Eine wichtige Technik für die Einsparung von Spuren ist die Erstellung eines Schlagzeug-Tracks, der möglichst alle Percussion- und Drumsounds überzeugend wiedergeben können soll. Wie jedoch oben bereits erwähnt, sind sämtliche Sounds nur bis zum Einsetzen der nächsten Note zu

hören, da eine Stimme immer nur ein Instrument gleichzeitig wiedergeben kann. Da Schlagzeug- und Percussion-Sounds so gut wie immer vom One-Shot-Typ sind, sollte man das Hauptaugenmerk auf die Länge der benötigten Instrumente richten. Nur wenn ein Sound eine wirklich beträchtliche Länge hat, wie z.B. ein Ride-Becken, oder wenn unbedingt zwei Instrumente gleichzeitig ertönen müssen, sollte man eine weitere Spur damit belegen.

Für die Erstellung eines Schlagzeug-Tracks geht man am einfachsten so vor:

❑ Zuerst wird der Basis-Beat kreiert. Dazu erstellt man meist nur einen einzigen Takt, in dem der Grundrhythmus festgelegt wird. Mit Hilfe der Repeat-Funktion kann das Ergebnis laufend kontrolliert werden. In diesem Beispiel wird auf der Grundlage einer Basedrum und einer Snare ein einfacher Rock-Rhythmus geschaffen:

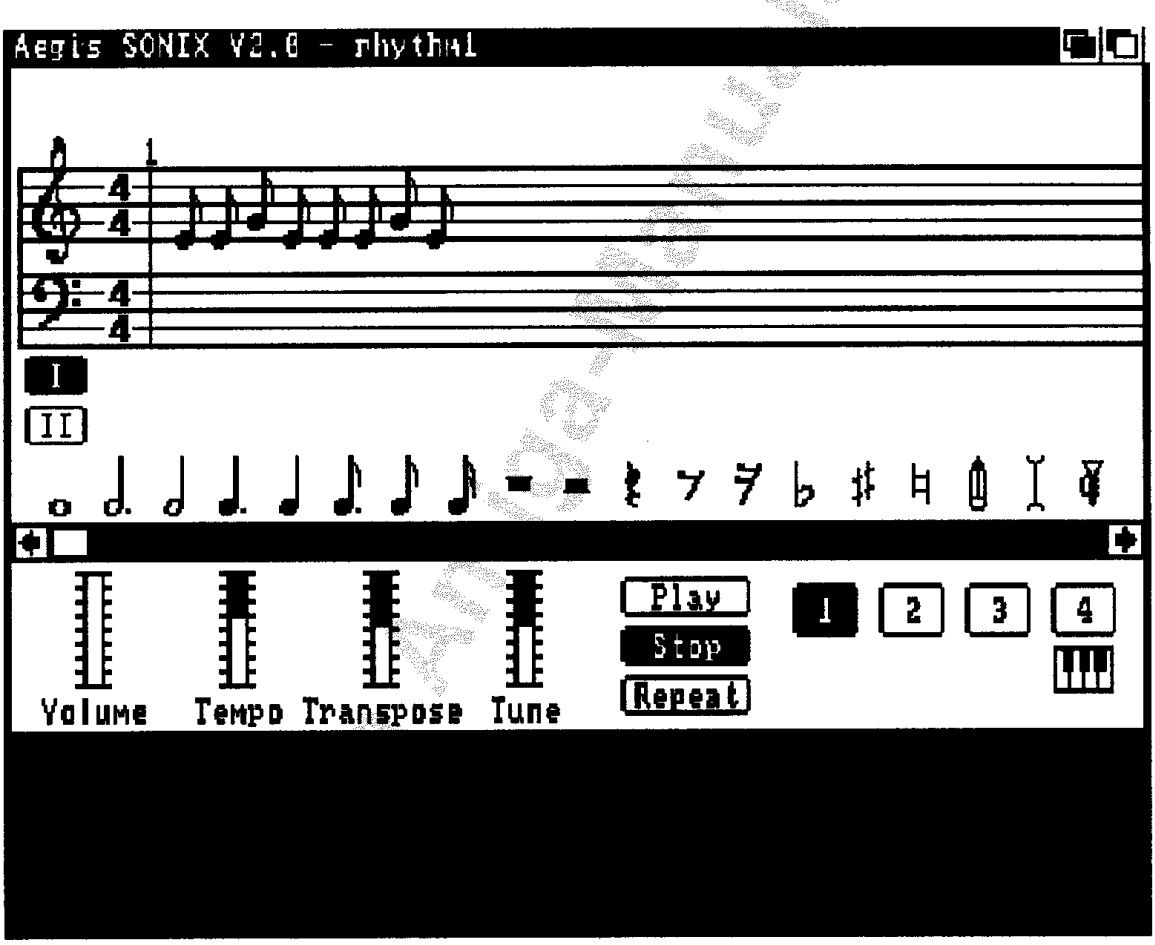


Abb. 4.1: Der Basis-Beat

❑ Als eine Variation könnte nun z.B. einfach der erste Beat in jedem zweiten Takt weggelassen werden. Dazu wird zunächst der ganze Takt markiert und die Copy-Funktion angewählt, um ihn dann mit Hilfe der Paste-Funktion zu duplizieren. Nun verwandelt man die letzte Note von Takt 1 von einem Sechzehntel in ein Achtel und löscht die erste Achtelnote in Takt 2, um einen vorgezogenen Beat zu erzielen:

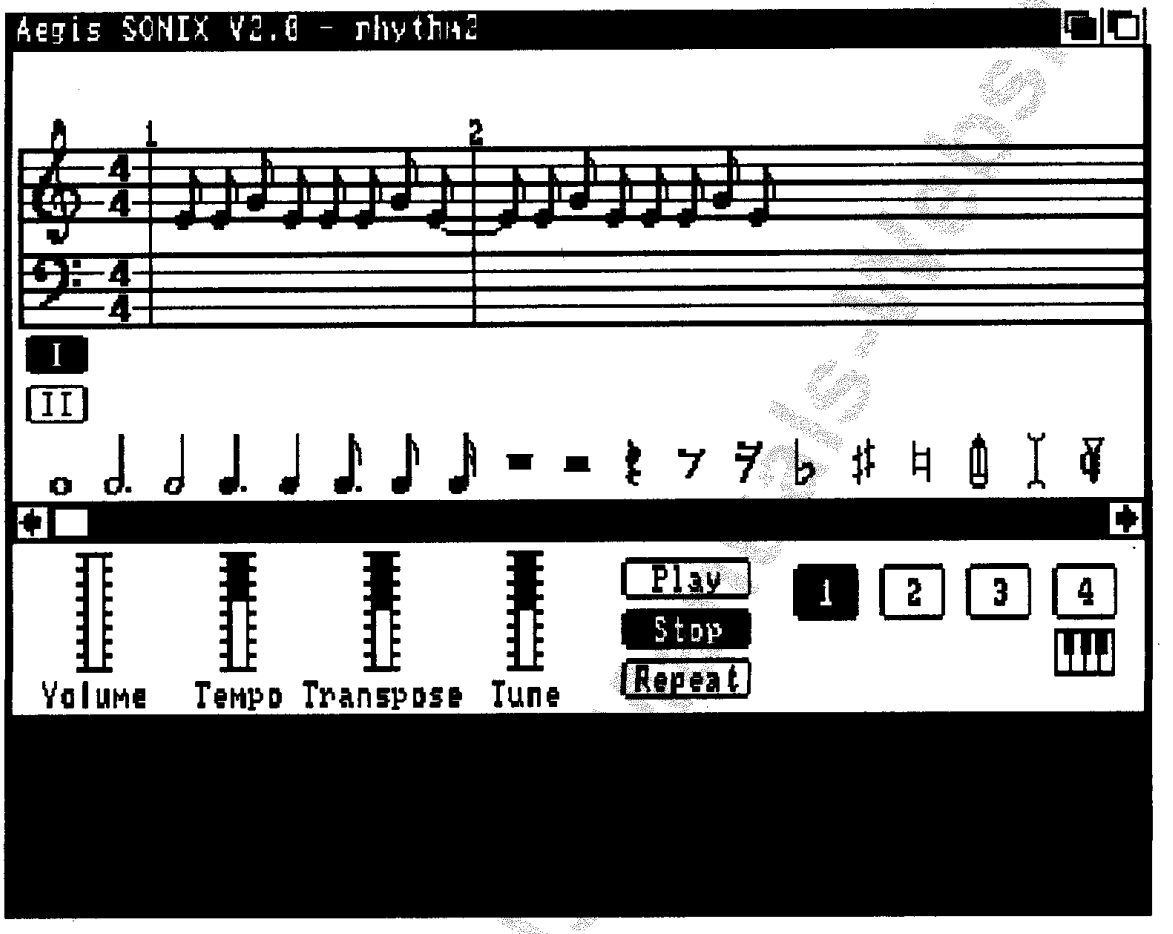


Abb. 4.2: Erweiterung des Beats auf zwei Takte

q Wenn einem das Ergebnis tauglich erscheint, so sollte man bereits darangehen, die ersten Passagen des Stückes, das man gerade im Kopf hat, vollständig niederzulegen. Während dieser Arbeit ergeben sich die meisten der weiteren Variationen, die für ein abwechslungsreiches Stück notwendig sind, so gut wie von selbst. Man könnte z.B. jeden vierten Takt so enden lassen, wie in Abbildung 4.3 gezeigt.

4.3.3 Backing-Tracks

Mit den vier Audio-Kanälen des Amiga lassen sich bekanntlich nicht nur monophone Instrumente, sondern jeder beliebige Sound abspielen, der mit den verfügbaren Sampling-Raten aufgenommen und befriedigend wiedergegeben werden kann. Dadurch ergeben sich klanglich enorme Möglichkeiten: Ganze Passagen, wie z.B. Schlagzeug-Rhythmen, Gitarren-Riffs und vieles mehr können in ein einziges Instrument gepackt und bei Bedarf in voller Länge abgespielt werden. Vor allem mit Hilfe der Loop-Option lassen sich sehr schön konstante und beeindruckend volle Begleitstimmen konstruieren.

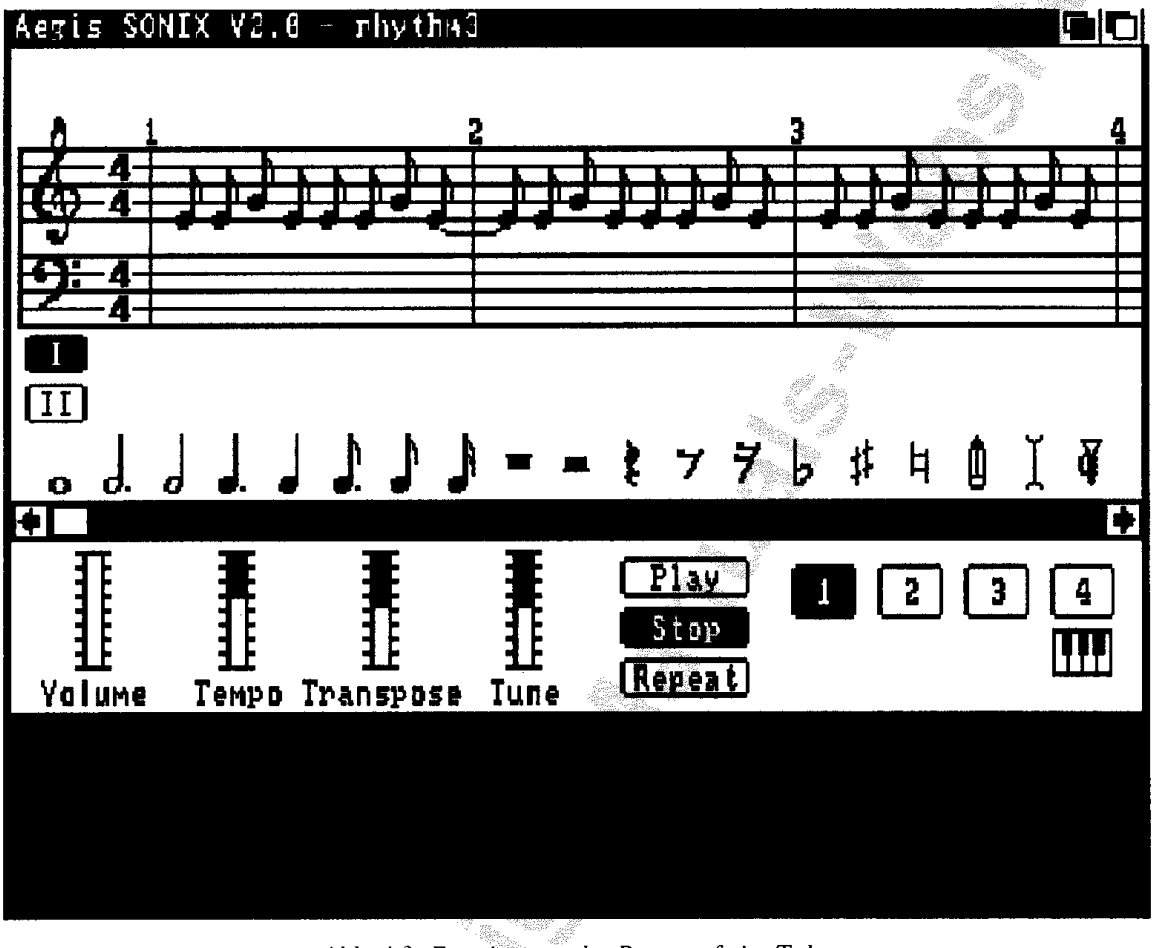


Abb. 4.3: Erweiterung des Beats auf vier Takte

Als erster Schritt zu einem solchen Backing-Instrument muß zunächst eine geeignete Soundpassage gefunden und aufgenommen werden. Auch hier muß wieder die Soundquelle auf die Standard-Samplingraten gestimmt werden. Am besten verwendet man für diesen Zweck die höchstmögliche Rate, 15838 Hz. Kann man die Tonquelle nicht stimmen, so muß wie gesagt die Sampling-Rate entsprechend korrigiert werden. Bei dieser Frequenz würde der Tuning Tone des Audiomaster ein H wiedergeben, worauf dieser dann genau auf den Sound abgestimmt und die richtige Sampling-Rate abgelesen werden kann. Es kann aber sein, daß ein H zum Stimmen ungeeignet ist, da es in der Tonleiter des Sounds nicht enthalten ist und deshalb nie harmonisch klingt. Dann muß der Pitch-Regler zuerst auf einen passenden Ton gestellt, mit dem Fine-Regler genau gestimmt und dann wieder auf das H zurückgegangen werden. Die Anzeige enthält nun die korrigierte Sampling-Rate für die Wiedergabe auf dem H.

In den meisten Fällen wird man wohl ein konstantes Instrument bevorzugen, so daß man eine möglichst einfach zu loopende Passage wählen sollte. Das Tempo des gesampleten Sounds gibt nun das Tempo des gesamten Stückes vor; der Temporegler des Kompositionsprogrammes muß sehr genau darauf eingestellt werden. Möchte man noch weitere Backing-Tracks in den Score

integrieren, so sollte man darauf achten, daß diese möglichst auf das Byte genau die gleiche oder ein Vielfaches der Länge des ersten aufweisen, damit das Timing garantiert an allen Stellen korrekt ist.

Hat man ein geeignetes Sample erstellt und ggf. geloopt, so speichert man es nun als einoktavige Wellenform ab; für die Verwendung in D.M.C.S. als IFF-Instrument, für Sonix lieber im dort vorgesehenen Sampled-Sound-Format. Bei Sonix ist es allerdings noch zusätzlich nötig, das Release-Level (Level 4) der Hüllkurve auf den höchsten Wert zu setzen, um eine konstante Wiedergabe zu ermöglichen. Nun aktiviert man das Instrument auf einer Spur und setzt einige ganze Noten an deren Beginn. Abhängig davon, wie viele Beats der Sound dauert, können auch größere oder kleinere Notenwerte verwendet werden. In Sonix muß man längere Notenwerte wie gesagt mit Hilfe der Hüllkurve und eingefügten Pausen simulieren. Jetzt startet man die Wiedergabe mit der Repeat-Funktion und stellt das Tempo so ein, daß die Länge des Samples sich mit den gespielten Notenwerten genau deckt. Reicht die Auflösung, d.h. die Wertabstufung des Tempo-Reglers nicht aus, um eine vollkommene Deckungsgleichheit des Samples mit dem Timing herzustellen, so kann dies mit dem Feinstimmungs-(Tune-)Regler nachkorrigiert werden.

Taken from Amiga-Manuals.Net

MIDI 5

Jeder musikbegeisterte Amiga-Anwender wird schon irgendwann einmal an die Möglichkeit gedacht haben, seinen Amiga für Aufnahmen (oder auch Livekonzerte und Sessions) zu benutzen, egal ob nun im Amateur- oder im Profibereich. Da besteht zuerst einmal die Möglichkeit, den Amiga als Musikinstrument, sprich als Sampler einzusetzen. Wie in den letzten Kapiteln gezeigt wurde, unterliegt der Amiga zwar einigen Beschränkungen hinsichtlich Auflösung und Sampling-Rate, ist aber trotzdem sehr wohl in der Lage, mit einer separaten Filterung professionell klingende Sounds zu erzeugen. Die dafür nötigen Voraussetzungen wurden hier ja bereits eingehend besprochen.

Für einen Computer wie den Amiga gibt es jedoch in einem Home- oder Profistudio noch eine wesentlich ruhmvollere Aufgabe zu bewältigen: die Steuerung von anderen Musikinstrumenten über die sog. MIDI-Schnittstelle (MIDI = Musical Instrument Digital Interface). Diese stellt ein genormtes Verfahren zur Kommunikation zwischen Musikinstrumenten dar, zu dem heutzutage zumindest alle elektronischen Instrumente wie Keyboards, Synthesizer, Sampler und Schlagzeugmaschinen kompatibel sind. Für die meisten Instrumentalisten gibt es inzwischen speziell angepaßte sog. MIDI-Controller, allen voran natürlich das Keyboard, das bis heute das Standard-MIDI-Eingabemedium darstellt. Aber auch einem Gitarristen oder einem Schlagzeuger oder sogar einem Sänger ist es so möglich, den von ihm gespielten Ton (mit den Parametern Tonhöhe, Länge, Dynamik und anderen) an ein externes, MIDI-kompatibles Gerät weiterzugeben, das diesen erklingen läßt. Zum Beispiel könnte ein Saxophonist durch einen an sein Instrument angepaßten MIDI-Controller seine Melodien an einen Synthesizer schicken, der diese dann mit Trompetensound wiedergibt.

Diese durch die MIDI-Schnittstelle übertragenen Informationen lassen sich aber nicht nur zur direkten Steuerung von anderen Instrumenten einsetzen, sondern auch aufnehmen, speichern, schneiden und archivieren. Genau dies ist auch die Aufgabe, die der Computer in einem MIDI-System erledigt. Ihm wird die Rolle des Steuergerätes (auch Sequenzer genannt) übertragen, der andere ihm untergeordnete Geräte (Slaves) zur Wiedergabe der zum jeweiligen Zeitpunkt gewünschten Sounds veranlassen kann. Die Eingabe (Editing) der Notendaten kann hierbei auf zwei Arten erfolgen: Entweder können Melodien und Akkorde über die oben erwähnten MIDI-Controller vom Instrument direkt eingespielt oder am Computer durch Tastatur- oder Mauseingabe programmiert werden. Welche von beiden Möglichkeiten man im jeweiligen Fall vorzieht, hängt von der Arbeitsweise und nicht zuletzt von den instrumentalen Fähigkeiten des Musikers ab. Die MIDI-Schnittstelle ermöglicht somit unter anderem sogar die völlige Unabhängigkeit von einer konventionellen Mehrspur-Bandmaschine, sofern sämtliche beteiligten Instrumente MIDI-Daten senden und ihre Klänge auch über MIDI wiedergegeben werden können (was bei akustischen Instrumenten nicht geht).

Da der Amiga mit einer seriellen Schnittstelle ausgerüstet ist, steht seiner Verwendung als MIDI-Steuercomputer in einem professionellen oder Home-Studio nichts im Wege. In diesem Kapitel soll zuerst auf die MIDI-Schnittstelle selbst eingegangen werden, wobei sowohl die praktische Anwendung, sprich Anordnung und Verkabelung von MIDI-Hardware, als auch ihre Programmierung Schwerpunkte sein werden. Anschließend werden einige MIDI-Geräte vorgestellt.

5.1 Was ist MIDI?

1983 wurde mit der MIDI-Spezifikation 1.0 der erste internationale Standard für die Kommunikation zwischen Musikinstrumenten vorgestellt. Seither haben sich die meisten Hersteller elektronischer Instrumente auf diesen geeinigt, nicht MIDI-kompatible Keyboards zumindest sind so gut wie nicht mehr zu verkaufen. Um den hardwaremäßigen Aufwand so gering wie möglich zu halten, wurde für MIDI eine serielle Datenübertragung gewählt, bei der die einzelnen Bits nacheinander durch eine einzige Leitung geschickt werden. Das Prinzip der seriellen Übertragung läßt sich in Kürze etwa so beschreiben:

Zuerst einmal müssen sich beide Kommunikationspartner, Sender und Empfänger, über die Übertragungsgeschwindigkeit einig sein, hier 31250 Bit/s, also ein zeitlicher Abstand zwischen den Bits von 32 µs. Ist das der Fall, kann mit der Übertragung begonnen werden. Hierbei setzt der Sender die Leitung zuerst auf 0, wodurch die Uhr des Empfängers synchronisiert wird (Startbit). Dann sendet er nacheinander die zu übertragenden Bits, die vom Empfänger anhand des durch Übertragungsrate und Startbit vorgegebenen Zeitrasters interpretiert werden. Darauf folgen ein oder zwei sog. Stoppbits (immer 1). Dieser Vorgang wird wiederholt, so lange Daten zum Senden bereitstehen.

5.1.1 Die Interface-Hardware

Technisch gesehen ist ein MIDI-Interface nichts anderes als eine serielle Schnittstelle (RS232, asynchron), die mit 8 Datenbit, 1 Stoppbit, ohne Parität (8N1) und bei einer Übertragungsrate von 31250 Bit/s arbeitet. Diese wurde so hoch gewählt, um die zwangsläufige Verzögerung zwischen dem Absenden einer MIDI-Information (z.B. dem Drücken einer Taste am Controller) und ihrer Interpretation (dem Erklingen des Tones) so kurz wie möglich zu halten. Bei 8 Datenbit plus je ein Start- und Stoppbit = 10 Bit ergibt sich eine Übertragungszeit von 320 µs pro Byte. Da im MIDI-Protokoll mindestens zwei Byte für einen Ton nötig sind, bedeutet das bei der gleichzeitigen Übertragung von mehr als vier oder fünf Tönen schon eine Verzögerung von über 30 ms, was bereits als störend empfunden wird. Aus diesem Grund erwägt man die Verdoppelung der Baud-Rate auf 62500 Bit/s. (Zehn Stimmen sind in einem sequenzerorientierten MIDI-System durchaus keine Ausnahme).

Bezüglich der Kabel- und Steckerbelegung wurde hier ein besonderer Weg beschritten: Sende- und Empfangsleitung werden nicht in einem Kabel untergebracht, sondern sind getrennt auf zwei Buchsen verfügbar. Diese sind als 5-Pol-Din-Buchsen (180 Grad) ausgeführt. Der Standard ist allerdings nicht unbedingt optimal gewählt, da Stecker und Buchsen im harten Studio- und Road-Alltag nur allzu leicht Schaden nehmen können (schwer einzustecken, leiern bald aus, werden leicht zertreten u.ä.).

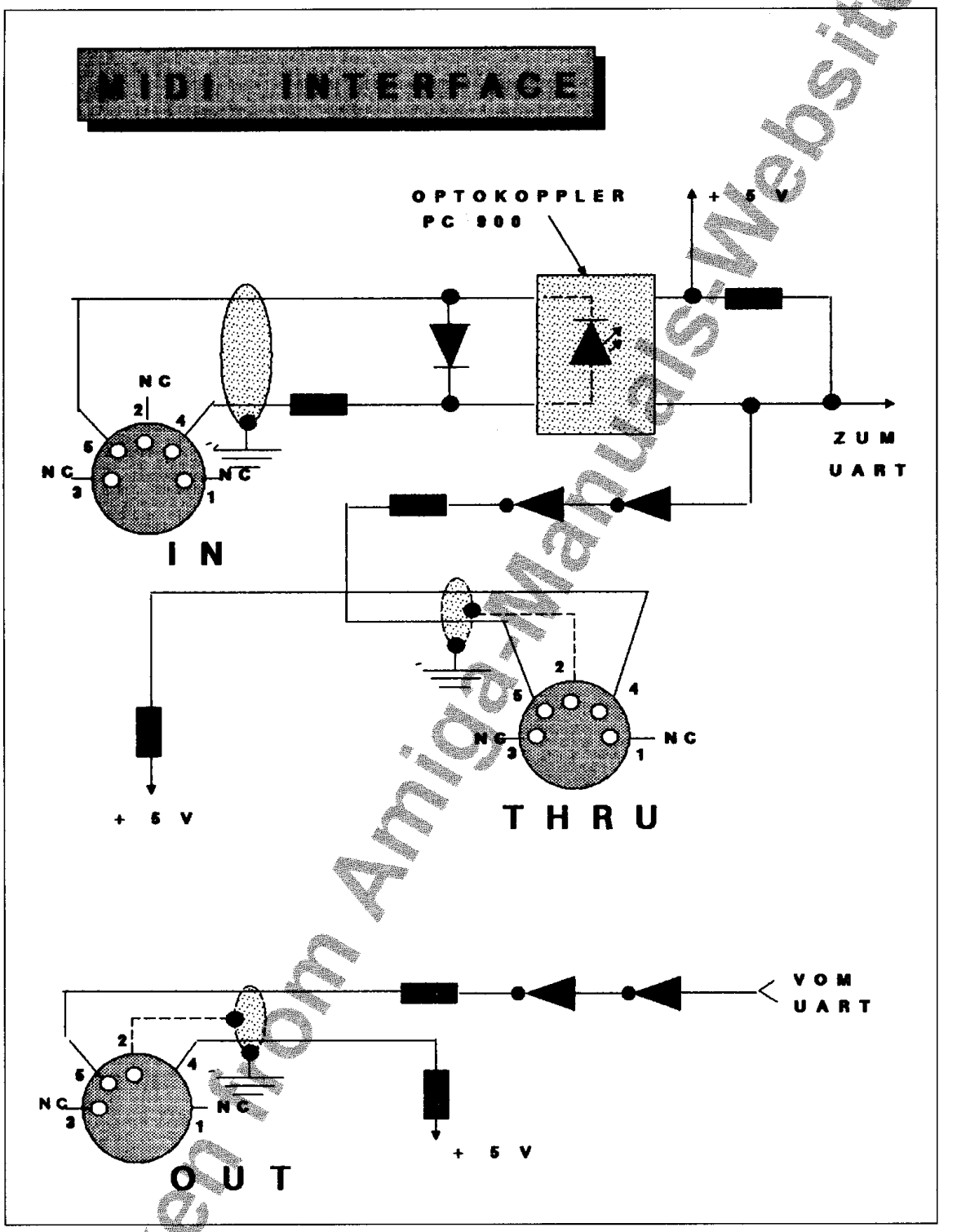


Abb. 5.1: Spezifikationsgerechte Schaltung des MIDI-Interface (Aus: Richard Aicher, Das Midi-Praxisbuch, 1988, Signum Medien Verlag)

Taken from Amiga-Manuals-Website

Die eine Leitung, auf der Daten gesendet werden sollen (MIDI OUT-Buchse), entspricht der TXD-Leitung der RS232 am Computer und muß auch im MIDI-Interface mit dieser verbunden werden. Die Empfangsleitung (MIDI IN-Buchse) entspricht RXD. Diese Buchse ist von der RXD-Leitung durch einen Optokoppler (lichtbetriebenen Schalter) galvanisch getrennt, um Brummschleifen zu vermeiden.

An den meisten MIDI-Interfaces befindet sich noch eine dritte Buchse, die MIDI THRU-Buchse. Hier werden alle an der Eingangsleitung ankommenden Signale direkt wieder ausgegeben, die MIDI IN-Buchse ist also nur auf diese Buchse durchgeschleift. Im Gegensatz zur RS232, die nur einen Kommunikationspartner ansprechen kann, können hier also theoretisch beliebig viele »Zuhörer« an eine MIDI OUT-Leitung angeschlossen werden. Deshalb spricht man auch vom MIDI-Bus (lat. omnes = alle; omniBUS = für alle).

In der Praxis gibt es hier jedoch das Problem, daß der Optokoppler aufgrund seiner Durchschaltdauer das Signal bei jedem Durchgang ein wenig verfälscht. Deshalb dürfen höchstens etwa drei MIDI-Geräte auf diese Weise in Serie geschaltet werden, da sonst die letzten Geräte in der Kette ein zu unsauberes Datensignal empfangen. Die hier auftretenden Übertragungsfehler können sehr tückisch sein, weil sie unter Umständen erst nach einiger Zeit auftreten, wenn sich die Bauteile erwärmt haben. (Das ist in einem Live-Gig wirklich super-peinlich!) Bei mehr als drei Geräten sollte man deshalb mit Hilfe einer sog. THRU-Box mehrere parallele Ausgänge schaffen, an denen jeweils nur ein oder zwei Geräte hängen (Sternschaltung, siehe Abbildung 5.3).

In Abb. 5.1 ist der normgerechte Aufbau des MIDI-Interface dargestellt. Es arbeitet mit einer 5-mA-Stromschleife (logisch 0 = Strom ein), die jeweils von MIDI OUT oder MIDI THRU mit Power versorgt wird. Hier ist u.a. der Optokoppler zu erkennen, der die Geräte galvanisch voneinander trennt. Da dieser mit einer Spannung zwischen zwei Leitungen geschaltet wird, gibt es für die eigentliche Datenübertragung ein Paar von zwei Adern (Pin 4 und 5), die in einem ordentlichen MIDI-Kabel miteinander verdrillt sein sollten (Twisted-Pair-Return). Ein solches Kabel darf nicht länger als 15 Meter sein. Die Abschirmung muß auf beiden Seiten an Pin 2 angeschlossen werden, kommt jedoch immer nur mit der Abschirmung der MIDI OUT- oder MIDI THRU-Buchse in Kontakt, da Pin 2 der MIDI IN-Buchse unbelegt ist. Pin 1 und 3 sind immer unbelegt.

5.1.2 MIDI-Schaltschemata

Die einfachste Art, via MIDI Musikinstrumente miteinander zu verbinden, besteht in der Hintereinanderschaltung der Geräte. Hierzu wird die MIDI OUT-Buchse des Controllers (z.B. Keyboard, auf dem die Töne gespielt werden) mit der MIDI IN-Buchse des ersten Slaves verbunden. Die MIDI IN-Buchse jedes weiteren Gerätes muß dann jeweils mit der MIDI THRU-Buchse des vorhergehenden verbunden werden. Diese Verschaltung kann dazu benutzt werden, jeden auf dem Masterkeyboard gespielten Ton gleichzeitig auf mehreren Slaves erklingen zu lassen, also »gestacked« zu spielen. Abb. 5.2 zeigt eine solche Anordnung.

Hier ist allerdings noch keine Möglichkeit zur Aufnahme und Speicherung der auf dem Controller gespielten MIDI-Informationen vorhanden. Dazu wird ein Sequenzer oder eben ein Computer benötigt, der die Rolle des Masters übernimmt. Dieser wird im allgemeinen direkt mit dem Controller verbunden, indem MIDI OUT des Computers mit MIDI IN des Controllerkeyboards

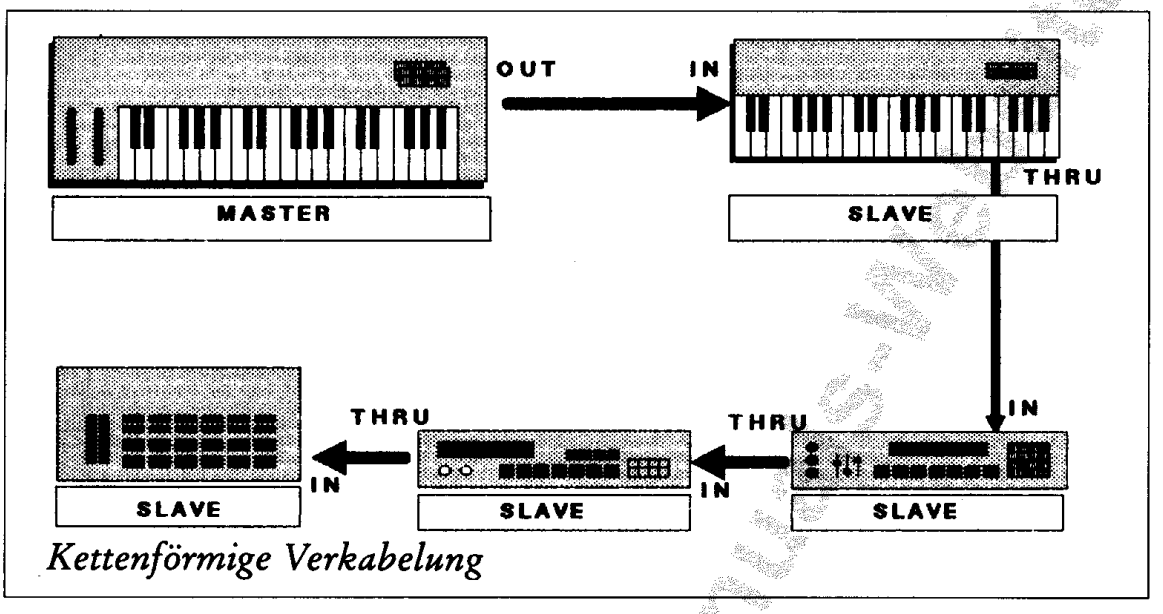


Abb. 5.2: Stacking mehrerer Slaves durch serielle Verschaltung (Aus: Richard Aicher, Das Midi-Praxisbuch, 1988, Signum Medien Verlag)

und umgekehrt, MIDI OUT des Controllers mit MIDI IN des Computers verbunden wird. Alle Slaves werden dann an der MIDI THRU-Buchse des Controllers angeschlossen (Abb. 5.3). Bei der Aufnahme, also dem Einspielen von Daten in den Computer, wird dessen MIDI IN-Eingang

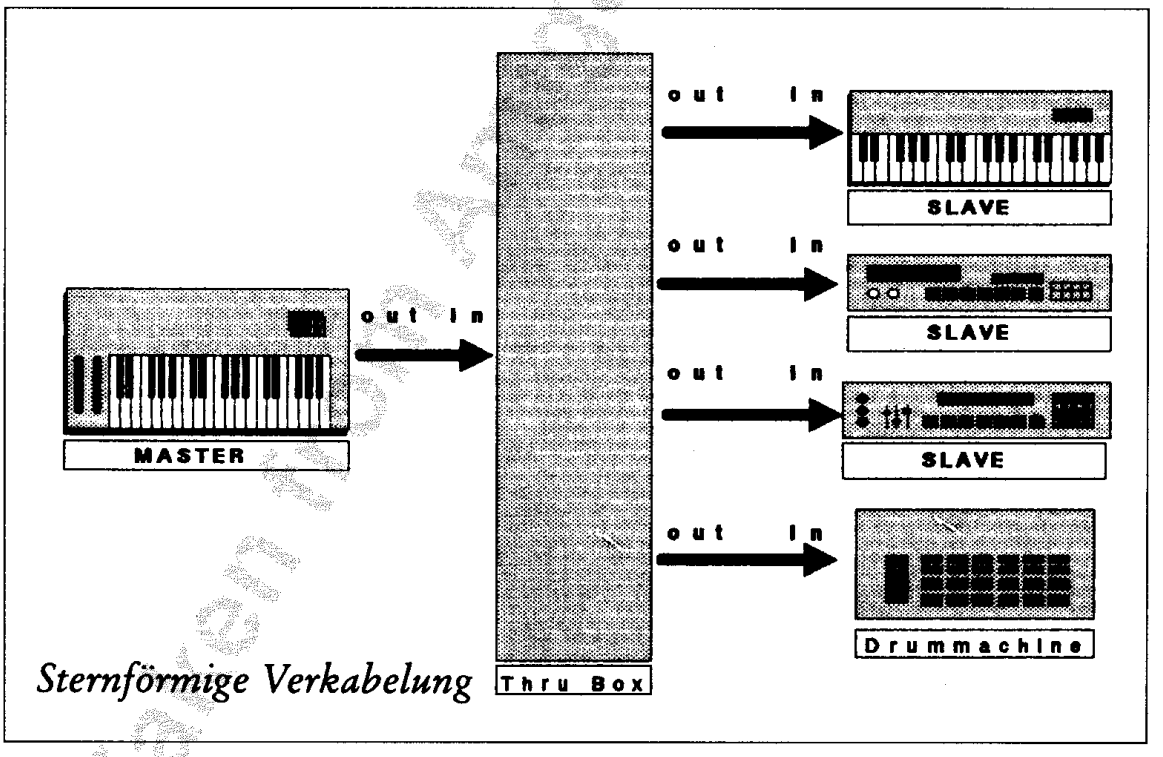


Abb. 5.3: Sternschaltung (Aus: Richard Aicher, Das Midi-Praxisbuch, 1988, Signum Medien Verlag)

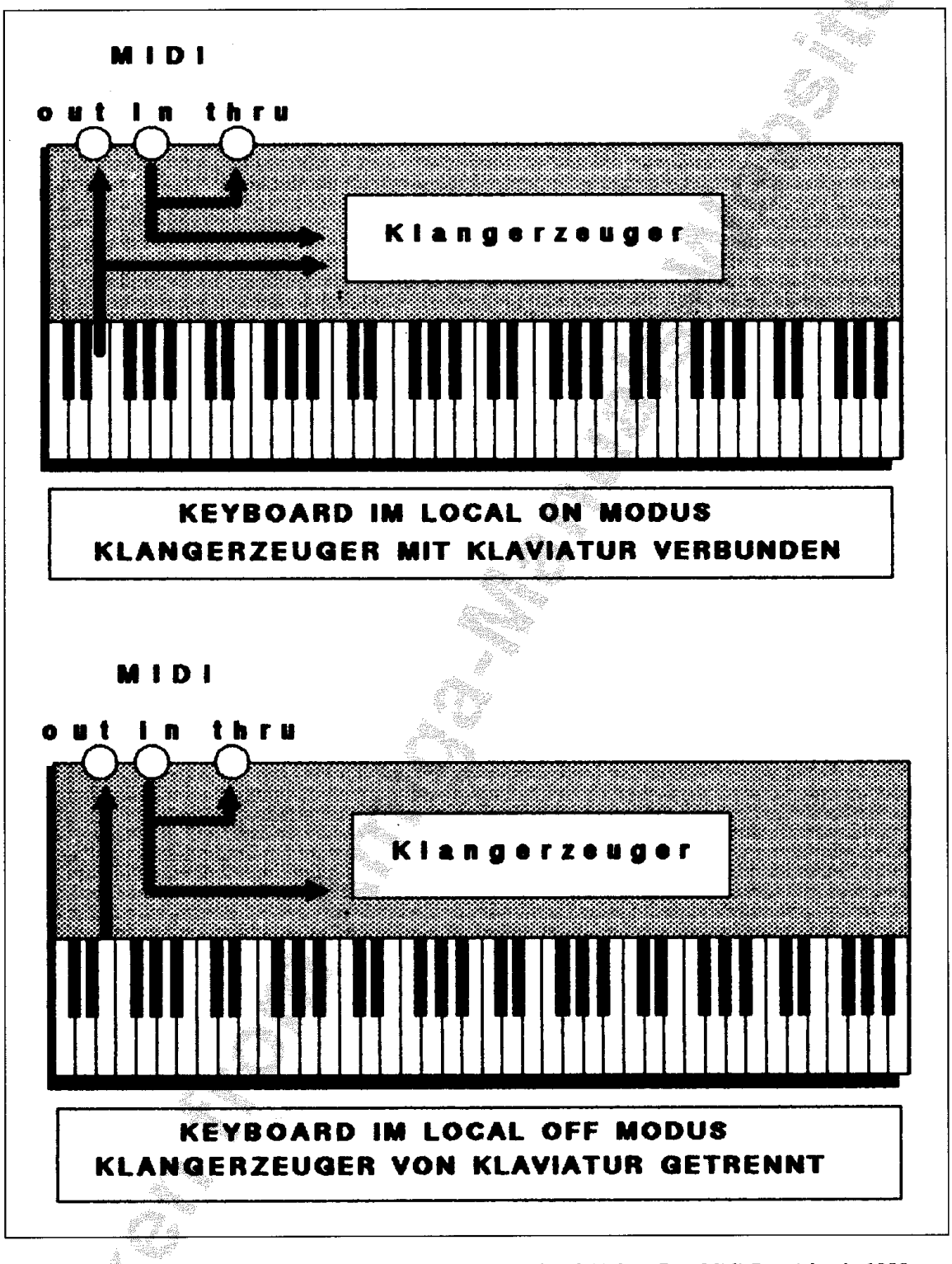


Abb. 5.4: Datenwege im Local-on/off-Modus (Aus: Richard Aicher, Das Midi-Praxisbuch, 1988, Signum Medien Verlag)

direkt wieder auf MIDI OUT gegeben, damit die auf dem Controller gespielten Noten auch von den Slaves wiedergegeben werden können. Bei der Wiedergabe, also beim Abspielen der im Computer gespeicherten Daten, gibt dieser die Daten wieder in den Bus und damit an die angeschlossenen Slaves. Natürlich müssen auch während der Wiedergabe weitere »Spuren« eingespielt werden können, um die klassische Mehrspur-Aufnahmetechnik, das Aufnehmen zu den bereits eingespielten Passagen, zu ermöglichen.

Wie oben erwähnt, gibt es jedoch beim seriellen Anschluß von mehr als drei Geräten Probleme, weshalb man sich nach einer Alternativlösung umsehen muß. Diese existiert in Form einer sog. THRU-Box, die den MIDI OUT-Anschluß eines Gerätes auf mehrere MIDI THRU-Buchsen verteilt. An dieser können dann an jedem Ausgang ein oder zwei Geräte angeschlossen werden, ohne fehlerträchtige Verzerrungen bei der Datenübertragung befürchten zu müssen. Dies bezeichnet man als Sternschaltung (Abb. 5.3).

Bei diesen Techniken ist es oft sinnvoll, wenn der Controller bestimmte Fähigkeiten aufweist, die die nähere Bestimmung des Zieles der MIDI-Daten ermöglichen. Eine dieser Fähigkeiten ist die sog. Local-on/off-Umschaltung. Diese erlaubt, die in einem Keyboard vorhandenen Tonerzeugungseinheiten von der Klaviatur zu trennen. Das ist gerade dann nötig, wenn man zur Begleitung der bereits eingespielten Spuren eine weitere aufnehmen will, die von einem anderen Slave als dem Keyboard selbst wiedergegeben werden soll. Hier kann im Local-off-Modus die neue Spur über die Klaviatur eingespielt werden, deren Daten über den Computer an das eigentliche Wiedergabegerät gehen. Der Tongenerator des Controllerkeyboards gibt dann nur die Töne wieder, die durch die MIDI IN-Buchse des Keyboards hereinkommen (Abb. 5.4).

Eine weitere wichtige Eigenschaft eines Controllers (im Fall eines Keyboards) ist das sog. Keyboard-Splitting. Hier kann die Klaviatur (oder allg. der Tonumfang des Controllers) in mehrere Bereiche aufgeteilt werden. Dies kann z.B. dazu eingesetzt werden, daß der untere Tonbereich (die linke Hand des Keyboarders) bis zu einem bestimmten Punkt mit dem Sound A, der obere (rechte Hand) mit Sound B wiedergegeben wird. Dies erlöst den Musiker von dem Berg von Eingabegeräten, die sonst für solche Anwendungen nötig wären.

Beim Kauf eines MIDI-Controllers ist also unbedingt darauf zu achten, daß zumindest diese beiden Voraussetzungen erfüllt sind, da sonst wichtige Möglichkeiten des MIDI-Systems nicht voll genutzt werden können.

5.1.3 Die MIDI-Channels (Kanäle)

Einer der wesentlichen Parameter für eine Information im MIDI-System ist die Angabe des Geräts, für das die Information bestimmt ist. Gäbe es keine Möglichkeit, z.B. an einen beliebigen Synthesizer in der Kette allein eine Note zu senden, so würden alle Slaves diese Note gleichzeitig wiedergeben. Dies würde zwar zu einer großen Klangfülle, nicht aber zur Realisation der von uns avisierten Mehrspur-Technik führen, wo ja einzelne Stimmen von bestimmten Geräten wiedergegeben werden müssen.

Abhilfe schafft hier nur die Methode, einer Botschaft auf dem MIDI-Bus eine »Hausnummer« des Gerätes mitzugeben, an das sie gerichtet ist. Diese Hausnummern bezeichnet man als MIDI-Channels oder MIDI-Kanäle. Für diese sind in jeder MIDI-Botschaft, die eine solche benötigt,

4 Bit reserviert, wodurch genau 16 verschiedene Kanäle zur Verfügung stehen. Jedem Controller kann im MIDI-System ein eigener Sendekanal (Send-Channel), jedem Slave ein eigener Empfangskanal (Receive-Channel) zugeordnet werden. Ein angeschlossener Slave empfängt zwar alle über den Bus laufenden Daten, aber interpretiert und führt nur diejenigen aus, die an dieses Gerät »adressiert« sind.

5.1.4 Die MIDI-Modes (Betriebsarten)

Nun müssen wir allerdings noch weitere Kriterien einführen, nach denen ein MIDI-Gerät die ankommenden Daten auswerten soll. Instrument ist nicht gleich Instrument und Musiker nicht gleich Musiker, die Anforderungen an die angeschlossenen Erweiterungsgeräte und ihre Arbeitsweise differieren daher stark. Soll wie gesagt ein angeschlossener Synthi nur die auf dem Masterkeyboard gespielten Noten wiedergeben, so werden z.B. keine Kanalangaben benötigt, der Synthi sollte sie ignorieren. Soll aber ein Keyboard-Split mit jeweils mehreren verschiedenen Sounds (unter Umständen aus verschiedenen Geräten) auf jedem Split durchgeführt werden, so sind Kanalinformationen in jedem Fall nötig.

Des weiteren werden beim Keyboard meist mehrere Töne zugleich gespielt, es ist ja ein voll polyphones Instrument. Die über einen Kanal gespielten Töne müssen deshalb an die Stimmen der Slaves verteilt werden, da jeder Synthi ja nur über eine begrenzte Anzahl verfügt. Der Nachteil dabei ist, daß jetzt alle anderen an diesen Kanal gerichteten Informationen, wie z.B. das Drehen des Modulationsrads, nun auf alle Töne wirken, die auf dem Kanal gespielt wurden. Ein MIDI-Gitarrist möchte aber z.B. gerne eine Saite hochziehen, während die anderen Töne unberührt bleiben sollen. Bei ihm wäre es also praktisch, jeder Saite einem eigenen Kanal zuzuordnen, auf dem immer nur ein Ton gespielt und dementsprechend moduliert wird; die Kanaldaten sollen also monophon behandelt werden. Diese Unterschiede in den Ansprüchen der verschiedenen Instrumentalisten waren es, die den Anstoß für die Implementation der MIDI-Modes gaben.

Die den momentanen Betriebszustand bestimmenden Parameter kann man sich als zwei Schalter vorstellen, von denen jeder zwei verschiedene Stellungen einnehmen kann. Der eine wird der »Omni on/off«-Schalter genannt. Dieser bestimmt, ob den Kanalnummern in den hereinkommenden MIDI-Botschaften überhaupt Beachtung geschenkt wird. Ist der Schalter in der Omni-on-Stellung, so wird jede Information ausgewertet, egal ob sie mit einer Kanalkennung versehen ist oder nicht.

Ist der Schalter dagegen in der Omni-off-Stellung, so wird zuerst geprüft, ob das Gerät (oder einige seiner Stimmen) diesem Kanal zugeordnet ist und auf ihm empfangen soll. Soll eine der Stimmen auf den Kanal reagieren, so wird die Note gespielt bzw. die entsprechende Modulation vorgenommen. Ist keine der Stimmen auf den Kanal ausgerichtet, so wird die Botschaft als ganzes ignoriert.

Der andere Schalter wird »Poly/Mono«-Schalter genannt. Er bestimmt, ob auf einem Gerät oder einem Kanal immer nur eine Note oder mehrere gleichzeitig gespielt werden können. Steht der Schalter auf Mono, so wird jede ankommende Note sofort wiedergegeben. War in dem Moment eine vorher auf dem Kanal gespielte Note noch nicht zu Ende, so wird diese gestoppt und ihr die Stimme »gestohlen«. Wenn also mehrere Töne kurz hintereinander auf einem Kanal eintreffen, so erklingt nur der Letzte.

Befindet sich der Schalter in der Poly-Stellung, so wird versucht, jede ankommende Note auf einer anderen Stimme im Gerät wiederzugeben. Es können natürlich immer nur so viele Töne gleichzeitig gespielt werden, wie Stimmen im Gerät vorhanden sind. Alle ankommenden Noten werden also über die vorhandenen Stimmen dynamisch verteilt. Übersteigt die Anzahl der Töne jedoch die der Stimmen, so wird dem zuerst gespielten Ton (und dann weiter in der Reihenfolge des Eintreffens) seine Stimme weggenommen und der neue Ton auf dieser gespielt.

Kombiniert man jeweils die zwei Stellungen der beiden Schalter, so erhält man insgesamt vier verschiedene Möglichkeiten, die im allgemeinen als die eigentlichen MIDI-Modes bezeichnet werden. Da nur drei der Kombinationen einigermaßen sinnvoll einzusetzen sind, wurden diese mit den Kürzeln Omni, Poly und Mono belegt:

Modus	**übliche Bezeichnung**
Omni on/Mono	–
Omni on/Poly	Omni
Omni off/Mono	Mono
Omni off/Poly	Poly

Omni on/Mono

Ein Synthesizer, der weder die ankommenden Daten nach Kanalnummern selektiert, noch mehrere Töne spielen kann, ist eine ziemlich eintönige Angelegenheit. Als (vollkommen monophoner) Synthesizer wäre ein solches Gerät wohl nur dann zu gebrauchen, wenn ihm ein MIDI-Kanaldatenfilter vorgeschaltet würde, so daß man noch andere Instrumente am Bus hinzufügen kann. Andernfalls würde er ja jede Note auf dem Bus versuchen mitzuplärren, und das auch noch monophon! Aus diesem Grund wird dieser Mode in der Praxis so gut wie nicht verwendet.

Omni on/Poly (Omni-Mode)

Dies ist der simpelste der verwendeten MIDI-Modes. Da auch hier keine Kanalnummern beachtet werden, spielt ein Gerät, das in diesem Mode arbeitet, alle am Bus auftretenden Noten (Omni on). Diese werden jedoch auf polyphone Weise auf die vorhandenen Stimmen verteilt, so daß zumindest so viele Noten gleichzeitig wiedergegeben werden können, wie Stimmen vorhanden sind. Werden jedoch mehr Noten empfangen, so werden wie erwähnt den zuerst angekommenen ihre Stimmen wieder gestohlen.

In diesem Mode wirken sich leider auch alle weiteren MIDI-Informationen auf alle Stimmen gleichzeitig aus, wie z.B. das oben erwähnte Modulationsrad. Deshalb können Geräte, die ausschließlich in diesem Mode arbeiten, wirklich nur in ganz einfache MIDI-Systeme integriert werden.

Omni off/Poly (Poly-Mode)

Der Poly-Mode ist der am häufigsten eingesetzte MIDI-Mode. In diesem Mode werden alle empfangenen Informationen zuerst einmal auf Übereinstimmung mit der am Gerät eingestellten

Kanalnummer geprüft (Omni off). Ist die Nummer richtig, so werden die Noten auf die freien Stimmen verteilt, wenn nicht, werden sie ignoriert. Normalerweise erklingt dann jede dieser Noten im selben Sound, also entweder alle nur nach Piano, Trompete etc.

Es gibt allerdings auch Slaves, die ihre Tonskala in zwei oder mehr Bereiche unterteilen und jedem von ihnen einen eigenen Sound zuordnen können. Solche Geräte werden auch Split-Synthesizer genannt. So könnte beispielsweise ein Sampler seinen Tonumfang mit mehreren Samples belegen, die alle nur eine Quint oder Quart auseinanderliegen, um die durch die Änderung der Ausgaberate bedingte Verfremdung so gering wie möglich zu halten. Auch könnte ein MIDI-Schlagzeuger seinen Trommeln lediglich verschiedene Notenwerte zuweisen, um sie dann auf einem einzigen Kanal an einen Slave zu schicken. Dieser würde dann z.B. bei einem C1 die Tom, bei einem Cis1 die Snare, bei einem D1 die Bassdrum wiedergeben, und so weiter. Diese Möglichkeiten machen den Poly-Mode schon für eine sehr große Zahl von Musikern interessant.

Omni off/Mono (Mono-Mode)

In diesem Mode arbeiten die meisten Sequenzer-orientierten Systeme, aber auch von MIDI-Gitarristen und MIDI-Schlagzeugern wird er bevorzugt. Auch hier werden die am Bus ankommenden Informationen auf die evtl. vorhandene Kanalnummer untersucht, und nur im Falle einer Übereinstimmung ausgewertet. Hierbei wird jedoch jeder einzelnen Stimme des Slaves eine eigene Kanalnummer zugeordnet. Die Auswertung der Noteninformationen findet jedoch nach monophonem Muster statt: Wird eine weitere Note auf dem Kanal gesendet, bevor die letzte zu Ende ist, so wird diese einfach abgeschnitten und die neue Note gespielt.

Den am Gerät eingestellten Empfangskanal nennt man in diesem Fall den Basiskanal. Diesem wird die erste verfügbare Stimme zugeordnet. Alle weiteren Stimmen werden dann den jeweils nächsthöheren Kanälen zugeordnet, wenn also der Basiskanal die Nummer n hat den Kanälen n+1, n+2, n+3 usw. Bestimmt man z.B. Kanal 3 als den Basiskanal, so wird Kanal 3 über Stimme 1, Kanal 4 über Stimme 2, Kanal 5 über Stimme 3 wiedergegeben etc.

Dieser Mode hat den großen Vorteil, daß jeder Stimme im Gerät sowohl ein eigener Sound als auch alle sonstigen kanalbezogenen Parameter exklusiv zugeordnet werden können. Dies war zumindest von den Vätern des MIDI-Systems so geplant. Viele Hersteller gingen hier jedoch einen einfacheren und weniger aufwendigen, aber unkorrekten Weg: Bei ihren Produkten werden Informationen wie Modulationsrad oder Soundumschaltung einfach auf alle Stimmen des Geräts bezogen, was speziell im Fall der Tonhöhenmodulation (man denke an den Gitarristen) natürlich vollkommen unsinnig ist. Beim Kauf eines Mono-Mode-Synthesizer-Slaves sollte man sich also vergewissern, daß dieser sämtliche kanalbezogenen Informationen auch den entsprechenden Kanälen zuordnen kann.

In den letzten Jahren hat sich unter den Herstellern noch eine Variation dieses Modes etabliert, weshalb man diesen nun in den Mono-Mode A und Mono-Mode B unterteilen muß. Der Mono-Mode A entspricht dabei der zuvor besprochenen Betriebsart: Kanal-Informationen werden selektiert, jedem Kanal wird eine Stimme zugeordnet.

Im Mono-Mode B werden die Noten im Gegensatz zum konventionellen Mono-Mode auch polyphon verarbeitet. Manche Geräte erlauben z.B. eine feste Verteilung der vorhandenen

Stimmen an die Kanäle. So könnte Kanal 1 etwa 2 Stimmen mit Sound A, Kanal 2 vielleicht 4 mit Sound B und Kanal 3 eine Stimme mit Sound C zugeordnet werden. Andere ermöglichen die dynamische Verteilung der Kanäle auf die vorhandenen Stimmen. Selbstverständlich sollten auch alle kanalbezogenen Informationen nur auf die jeweiligen Kanäle wirken, ansonsten könnte man ja gleich den Poly-Mode verwenden.

5.1.5 Das MIDI-Datenformat

Sämtliche MIDI-Informationen werden als Bytes, also als 8 Bit breite Binärdaten übertragen. Sie bestehen im allgemeinen aus einem sog. Status-Byte, das die Art des übertragenen MIDI-Befehls kennzeichnet, sowie einem oder mehreren Daten-Bytes, die die Parameter für diesen Befehl beinhalten. Diese beiden Grundtypen von Datenworten werden im MIDI-System aussschließlich anhand des höchstwertigen Bits, der Nummer 7, unterschieden:

Status-Byte: *1sss ssss* höchstwertiges Bit immer gesetzt (d.h. gleich 1);
der numerische Wert dieses Bytes ist also immer größer oder gleich 128.

Daten-Byte: *0ddd dddd* höchstwertiges Bit immer gelöscht (d.h. gleich 0);
der numerische Wert dieses Bytes ist also immer kleiner als 128.

Dies ermöglicht bei den meisten digitalen Prozessoren eine sehr schnelle Unterscheidung zwischen Status- und Datenbytes, da sich das sog. Negativ-Flag, das bei so gut wie bei allen Prozessoren vorgesehen ist, bei der Übernahme eines Wertes in ein Prozessorregister oder einer anderen Verarbeitung meist automatisch auf den Wert des höchstwertigen Bits einstellt.

Empfängt ein Gerät ein Status-Byte, so ändert das Gerät seinen Empfangsstatus, d.h., es erwartet von nun an die durch das Status-Byte implizierte Art von Daten. Dieser Zustand wird aufrechterhalten, bis ein Statusbyte mit anderem Inhalt empfangen wird. Das bedeutet, daß ein Statusbyte nur einmal gesendet werden muß, bzw. ihm auch mehrere Datengruppen des verlangten Inhalts folgen können, so lange nur die Anzahl der Daten-Bytes für jede Gruppe stimmt. Um z.B. mehrere Noten an einen Kanal auszugeben, genügt es, das Status-Byte 1001 nnnn (Note auf Kanal nnnn einschalten) und danach beliebig viele 2-Byte-Gruppen mit dem Inhalt 0kkk kkkk 0vvv vvvv (Noten-Nr., Anschlagsdynamik) zu senden. Sequenzer-Programme sollten aus den empfangenen Daten unbedingt die überflüssigen Statusbytes entfernen können, um Übertragungszeit einzusparen.

Eine Ausnahme machen hier die MIDI-Echtzeitbefehle. Diese bestehen nur aus einem Status-Byte, und haben deshalb auch keinen weiteren Effekt auf den Empfangsstatus. Echtzeitbefehle können auf diese Weise in jeden MIDI-Befehl eingefügt werden, so lange dieser mindestens 2 Byte lang ist.

Bei manchen Befehlen ist es notwendig, größere Werte zu übertragen, als es die Auflösung eines MIDI-Datenwortes (7 Bit = 128 Werte) erlauben würde. Hierzu werden diese Werte in ein höherwertiges (MSB) und ein niederwertiges Datenbyte (LSB) aufgeteilt. Da das 7. Bit jedoch nicht benutzt werden kann, da es ja immer 0 sein muß, kann der Gesamtwert nicht aus MSB und LSB direkt gewonnen werden, sondern muß zuerst in die 14-Bit-Binärdarstellung umgewandelt werden. Die einzelnen Bits werden folgendermaßen zugeordnet:

```
                    MSB                                                LSB
MIDI-Datenbytes      0  m  m  m   m  m  m  m      0  l  l  l   l  l  l  l
Bit-Nr.                 14 13 12  10 9  8  7         6  5  4   3  2  1  0
```

Um also ein 14-Bit-Format aus zwei MIDI-Datenwörtern zu erhalten, muß zunächst das Bit 0 des MSB nach Bit 7 der Binärdarstellung gebracht werden. Dann muß das MSB um eine Stelle nach rechts geschoben werden. Wenn man beide Ergebnisbytes hintereinander stellt, erhält man ein 14-Bit-Format, mit dem arithmetische Operationen durchgeführt werden können.

5.1.6 Die Unterteilung der MIDI-Befehle

Die im MIDI-System definierten Befehle lassen sich grob in zwei Klassen einteilen: kanalbezogene und nicht kanalbezogene (System-)Befehle. Die kanalbezogenen Befehle stellen sich aus den Stimm-Befehlen (Note-on = Ton an und Note-off = Ton aus) und den Mode-Befehlen (Angabe der Betriebsart) zusammen. Die nicht kanalbezogenen Befehle unterteilen sich in die Common- (betreffen alle Einheiten), die Echtzeit- (Start-, Stop- und Timersignale) und die Exclusiv-Befehle (enthalten Hersteller-spezifische Informationen). Hier noch einmal eine Übersicht:

```
                        MIDI-Befehle
                       (MIDI Messages)
                     /                 \
kanalbezogene Befehle                   System-Befehle
        /          \                   /          \          \
Stimme an/aus   Control Change      Allgemeine B.  Echtzeit-B.  Exklusiv
After-Touch     Mode Change         (Common)       (Real Time)
Pitch-Wheel     (Mode)
Prog Change
(Voice)
```

5.1.7 Die MIDI-Befehle auf einen Blick

Dieser Abschnitt stellt im wesentlichen eine Zusammenfassung der MIDI-Spezifikation 1.0 dar. Hier wurde versucht, die Unterteilung ein wenig günstiger und übersichtlicher zu gestalten, als es im amerikanischen Original der Fall ist. Zuerst folgt eine Tabelle der verschiedenen MIDI-Statusinformationen, nach denen der gesamte MIDI-Befehlssatz aufgegliedert ist. Darauf schließt sich eine genauere Erklärung eines jeden Statuswortes an, wobei die evtl. einem Statuswort untergeordneten Befehle (Mikro-Befehle) in dem jeweiligen Paragraph erläutert werden.

```
Status binär    hex    D.bytes    Erläuterung
-------------------------------------------------------------------------

        ; Channel Voice Messages (nnnn = Kanalnummer)

1000 nnnn       8n     2          NoteOFF - Event (Ton ausschalten)
1001 nnnn       9n     2          NoteON  - Event (Ton einschalten)
```

Status binär	hex	D.bytes	Erläuterung
1010 nnnn	An	2	Polyphonic Key Pressure (Druck auf eine Taste)
1011 nnnn	Bn	2	Control/Mode Change (Cntrl- u. Mode Änderung)
1100 nnnn	Cn	1	Program Change (Programmwechsel)
1101 nnnn	Dn	1	After Touch (Gesamtdruck auf alle Tasten)
1110 nnnn	En	2	Pitch Wheel Change (2. Modulationsrad)
; System Messages (sss, ttt = Message-Typ)			
1111 0000	Fx	0-?	System Exclusiv (Hersteller-spezifisch)
1111 0sss	Fx	0-2	System Common (gültig für das gesamte System)
1111 1ttt	Fx	0	System Real Time (Start,Stop,Timer,A.S.)

nnnn Kanalnummer - 1, d.h.
0000 entspricht Kanal 1
0001 entspricht Kanal 2
...
1111 entspricht Kanal 16
0sss siehe System Common-Befehle
0ttt siehe System Real Time-Befehle

1000 nnnn: Note off (Ton ausschalten)

Syntax: 1000 nnnn 0kkk kkkk 0vvv vvvv
8n Noten-nummer Dynamik

Beschreibung: Mit diesem Befehl wird ein mit dem Note-on-Befehl eingeschalteter Ton (siehe unten) wieder ausgeschaltet.

Parameter: 0kkk kkkk MIDI-Notennummer (0-127). Dieser Parameter bestimmt (zusammen mit der Kanalnummer) den Ton, der ausgeschaltet werden soll.
0vvv vvvv Loslaß-Dynamik. Bezeichnet die Geschwindigkeit, mit der z.B. am Keyboard eine Taste losgelassen wird, und damit Teile der Hüllkurve bzw. Filtereigenschaften. Wird leider nur von wenigen Geräten verarbeitet.

Anmerkungen:
Definitionsgemäß muß jedem Note-on-Befehl (siehe dort) nach einer gewissen Zeit ein Note-off-Befehl folgen. Der effektive Notenwert ergibt sich also aus der Zeitspanne zwischen Note-on- und Note-off-Befehl. Der Note-off-Befehl muß dieselbe Kanal- und Notennummer enthalten wie der entsprechende Note-on-Befehl.

1001 nnnn: Note on (Ton einschalten)

Syntax: 1001 nnnn 0kkk kkkk 0vvv vvvv
9n Noten-Nummer Dynamik

Beschreibung: Mit diesem Befehl wird ein Ton mit der Tonhöhe 0kkk kkkk und der Anschlagsdynamik 0vvv vvvv eingeschaltet.

taken from Amiga-Manuals-Website

Parameter: 0kkk kkkk MIDI-Notennummer (0-127). Da eine Oktave 12 Töne enthält, liegen oktavgleiche Töne 12 MIDI-Tonnummern auseinander. Dem mittleren C auf der Klaviatur entspricht dabei die Nummer 60. Ein 8oktavige Klaviatur reicht daher von 12 bis 108.

0vvv vvvv Anschlagsdynamik (0-127). Bezeichnet die Geschwindigkeit, mit der z.B. am Keyboard eine Taste angeschlagen wird, und damit Teile der Hüllkurve oder Filtereigenschaften. Die Skalierung sollte logarithmisch sein, also 127 = fff, 64 = mf, 1 = ppp, 0 = off.

Anmerkungen:
Definitionsgemäß muß jedem Note-on-Befehl nach einer gewissen Zeit ein Note-off-Befehl folgen (siehe dort). Alternativ kann ein Ton jedoch auch durch ein Note-on-Event mit der Anschlagsdynamik 0vvv vvvv = 0 ausgeschaltet werden (erspart eine Statusänderung nach Note-off).

1010: Aftertouch polyphon (mechanischer Druck auf eine Taste)

Syntax:	1010 nnnn	0kkk kkkk	0vvv vvvv
	An	Noten-Nummer	Druck-Wert

Beschreibung: Dieser Befehl gibt eine Änderung des mechanischen Drucks auf eine Keyboard-Taste an.

Parameter: 0kkk kkkk MIDI-Notennummer (0-127). Dieser Parameter bestimmt (zusammen mit der Kanalnummer) den Ton, dessen Dynamik im nachhinein verändert werden soll.

0vvv vvvv Druck-Wert (0-127). Bezeichnet den mechanischen Druck, der nach dem Anschlag auf die einzelne Taste gegeben wird.

Anmerkungen:
Dieser Befehl wird bei jeder registrierten Änderung des Tastendrucks um ein Intervall gesendet. Deshalb nehmen diese Informationen normalerweise sehr viel Übertragungszeit auf dem MIDI-Bus und Speicherplatz im Sequencer in Anspruch. In Systemen, wo der Polyphonic Aftertouch nicht benötigt wird, sollte er ignoriert bzw. gar nicht erst gesendet werden.

1011: Control/Mode Change (Regler- und Mode-Änderung)

Syntax:	1011 nnnn	0ccc cccc	0vvv vvvv
	Bn	Control-Nummer	Control-Wert

Beschreibung: Mit diesem Befehl können kontinuierliche Regler (z.B. Modulationsrad) und Schalter (z.B. Sustain-Pedal) sowie der Mode verändert werden. Die Funktion wird bestimmt durch die Control-Nummer 0ccc cccc:

0ccc cccc	Funktion
0-31	Control Change (Regler MSB)
32-63	Control Change (Regler LSB)
64-95	Switches (Schalter)
96-121	reserviert für zukünftige Anwendungen
122-127	Channel-Mode-Befehle (Betriebsart)

Control-Change (0-63)

Syntax:	1011 nnnn	0ccc cccc	0vvv vvvv
	Bn	Regler-Nummer	Regler-Wert (MSB oder LSB)

Parameter:	0ccc cccc	0vvv vvvv
	0	MSB (höherwertiges Byte) Regler 0
	1	MSB (höherwertiges Byte) Regler 1
	...	...
	31	MSB (höherwertiges Byte) Regler 31
	32	LSB (niederwertiges Byte) Regler 0
	33	LSB (niederwertiges Byte) Regler 1
	...	...
	63	LSB (niederwertiges Byte) Regler 31

Anmerkungen:
Mit Channel-Mode-Befehlen lassen sich der Local-on/off-Zustand sowie die verschiedenen Betriebsarten bestimmen. Des weiteren gibt es je einen Befehl zur Stummschaltung aller von einem Kanal erzeugten Töne und zur Initialisierung aller Control-Change-Meldungen. Der Reset-All-Controllers-Befehl (121, 79H) bewirkt, daß alle Controller wieder auf ihren DEFAULT-(Initialisierungs-)Wert gesetzt werden (z.B. Modulation=0, Panorama=64, usw.).

Anmerkungen:
Die Continous-Controller (kontinuierlich veränderbare Regler) sind bestimmt für analogähnliche Parameter, wie z.B. das Modulationsrad. Sie sind in ein höherwertiges (MSB) und ein niederwertiges Byte (LSB) aufgeteilt. Der resultierende Wertebereich entspricht 2*7 = 14 Bit, also von 0 bis 16383. Die MSB der einzelnen Regler (Nummer 0-31) werden mit der Regler-Nummer selbst, die LSB mit der Regler-Nummer plus 32 in 0ccc cccc angegeben. Diese Angaben werden im Prinzip bei jeder Änderung des jeweiligen Reglers gesendet, jedoch mit einem gewissen zeitlichen Mindestabstand, um den MIDI-Bus nicht zu überlasten. Wird das LSB nicht benötigt, da die Auflösung des Reglers nicht mehr als 128 verschiedene Werte beträgt, so wird nur das MSB gesendet. Hat sich bei einem hochauflösenden Regler nur das LSB verändert, so braucht nur dieses gesendet zu werden.
Da wie erwähnt der Empfangsstatus nur einmal durch ein Statusbyte festgelegt werden muß, braucht dieses bei mehreren Control Change-Befehlen nicht wiederholt zu werden.

Takt vom Amiga-Manuals-Website

Für die Angabe von MSB und LSB eines Reglers braucht also nur das Statusbyte 1011nnnn und dann 0ccccccc 0mmmmmmm (Reglernummer = 0-31, MSB) und 0ccccccc 01111111 (Reglernummer+32 = 32-63, LSB) gesendet zu werden.
Die Regler-Nummern sind in der MIDI-Spezifikation keinen bestimmten Funktionen zugeordnet (bis auf 1 = Modulationsrad). Hier wird auf die Dokumentation der jeweiligen Geräte verwiesen. Es haben sich jedoch mittlerweile folgende Konventionen entwickelt:

Definierte Regler-Nummern:

Nummer	Parameter	Wertebereich
1	Modulationsrad	0-127
2	Breath-Regler	0-127
4	Fußpedal-Regler	0-127
5	Portamento-Zeit	0-127
6	Data-Entry	0-127
7	Main-Volume	0-127
8	Balance	0-127
10	Panorama	0-127

Außerdem verwendete Regler-Nummern:

Nummer	Parameter	Hersteller
2	VCF-Modulation	Korg
4	Volume	Roland
4	Dynamik-Control	Roland
6	Master-Tune	Casio
9	Panning	Yamaha
11	Expression-Pedal	Roland
31	Pitchbend-Range	Roland

Die neueste Definition eines MIDI-Controllers (Stand Juni '90) betrifft das Umschalten der einzelnen Bänke (Gruppen von Programmspeicherplätzen, Sounds u.ä.) eines angeschlossenen MIDI-Geräts. Durch den Program-Change-Befehl war es bisher ja nur möglich, zwischen 128 unterschiedlichen Speicherplätzen umzuschalten. Da der Trend aber inzwischen dahingeht, mehr als 128 Speicherplätze pro Bank für Sounds/Samples/Effekteinstellungen usw. bereitzustellen, wurde von der IMA ein BANK-SELECT-Controller (Regler 0) definiert. Dieser erlaubt es nun, auch zwischen ganzen Gruppen von Programmen umzuschalten. Der neue Controller wurde erstmals in der KORG »Wavestation« implementiert.

Die Definition lautet:

Nummer	Parameter
0 (00H)	Bank-Select (MSB)
32 (20H)	Bank-Select (LSB)

Taken from Amiga-Manuals-Website

Eine Bank-Select-Meldung muß immer als Paar übertragen werden und erlaubt als 14-Bit-Wert 16384 verschiedene Einstellungen.

Ein Beispiel: n = MIDI-Kanal

Status	Data 1	Data 2	
Bn	0 (00H)	0 (00H)	(Bank-Select MSB)
Bn	32 (20H)	128 (7FH)	(Bank-Select LSB)

Dieser MIDI-Befehl (nacheinander übertragen) schaltet auf die 128. Bank eines angeschlossenen MIDI-Geräts.

Switches (64-95)

Syntax:	1011 nnnn	0ccc cccc	0vvv vvvv
	Bn	Schalter-Nummer	an/aus
Parameter:	0ccc cccc	0vvv vvvv	
	64	0 = Schalter 0 off, 127 = on	
	65	0 = Schalter 1 off, 127 = on	
	...	...	
	95	0 = Schalter 31 off, 127 = on	

Anmerkungen:
Die Switches (Schalter) sind für Parameter bestimmt, die nur die Zustände »ein« oder »aus« einnehmen können, wie z.B. das Sustain-Pedal am Keyboard oder Klavier. In 0ccccccc steht dabei die Nummer des Schalters plus 64. Eine 0 in 0vvvvvvv bedeutet Schalter ein, 127 bedeutet Schalter aus. Switch-Befehle mit einem Wert von 1 bis 126 in 0vvvvvvv werden ignoriert.
Bei den Schaltern haben sich folgende Zuordnungen etabliert:

0ccccccc	Parameter	Wertebereich
64 (40H)	Hold/Sustain-Pedal	0=aus, 127=ein
65 (41H)	Portamento on/off	0=aus, 127=ein
66 (42H)	Sostenuto-Pedal	0=aus, 127=ein
67 (43H)	Soft-Pedal	0=aus, 127=ein
69 (45H)	Hold2-Pedal	0=aus, 127=ein
91 (5BH)	External Depth	0-127
92 (5CH)	Tremolo Depth	0-127
93 (5DH)	Chorus Depth	0-127
94 (5EH)	Celeste Depth	0-127
95 (5FH)	Phaser Depth	0-127
96 (60H)	Data Increment	0=aus, 127=ein
97 (61H)	Data Decremement	0=aus, 127=ein

Channel-Mode-Befehle (121-127)

Taken from Amiga-Manuals-Website

Syntax : 1011 nnnn 0ccc cccc 0vvv vvvv

Bn Mode-Befehl (siehe unten)

Parameter :

0ccc cccc	0vvv vvvv	Befehl
121 (79H)	0	Reset All Controllers
122 (7AH)	127	Local ON
	0	Local OFF
123 (7BH)	0	All Notes OFF (A.N.O)
124 (7CH)	0	Omni OFF/ A.N.O.
125 (7DH)	0	Omni ON/ A.N.O.
126 (7EH)	Kanäle	Mono ON(Poly OFF)/ A.N.O. Alle Stimmen ab Basiskanal zuordnen
127 (7FH)	0	Mono OFF(Poly ON)/ A.N.O.

Anmerkungen:
Mit Channel-Mode-Befehlen lassen sich der Local-ON/OFF-Zustand sowie die veschiedenen Betriebsarten bestimmen. Des weiteren gibt es je einen Befehl zur Stummschaltung aller von einem Kanal ereugten Töne und zur Initialisierung aller Control-Change-Meldungen. Der Reset-All-Controllers-Befehl (121, 79H) bewirkt, daß alle Controller wieder auf ihren DEFAULT-(Initialisierungs-)Wert gesetzt werden (z.B. Modulation=0, Panorama=64, usw.).
Der Local-on/off-Befehl (122) wird dazu benutzt, um die interne Verbindung zwischen Klaviatur und Klangerzeuger eines Instruments zu unterbrechen. Wird in 0vvvvvvv eine 0 empfangen, wird die Verbindung unterbrochen; die von der Klaviatur gesendeten Daten gehen nun nur noch über MIDI OUT an externe Slaves. Der interne Klangerzeuger erhält sie nicht mehr, er empfängt seine Informationen nur noch über die MIDI IN-Buchse des Instrumentes. Steht in 0vvvvvvv der Wert 127, so werden Klangerzeuger und Klaviatur wieder verbunden.
Der All-Notes-off-Befehl (123) wird dazu benutzt, alle Töne stummzuschalten. Da ja alle Channel-Mode-Befehle mit einer Kanalnummer im Status versehen sind, ist es von der Betriebsart abhängig, ob ein ganzes Gerät oder nur einige Stimmen abgeschaltet werden.
Der Omni-on-Befehl (124) schaltet den Omni-Mode ein (siehe 5.1.4). In 0vvvvvvv steht hierbei immer eine 0.
Der Omni-off-Befehl (125) schaltet den Omni-Mode aus (siehe 5.1.4). In 0vvvvvvv steht hierbei immer eine 0.
Der Mono-on-Befehl (126) schaltet den Mono-Mode ein bzw. den Poly-Mode aus (siehe 5.1.4). In 0vvvvvvv steht hierbei die Anzahl M der Kanäle, denen die Stimmen des Gerätes zugeordnet werden sollen. Dem Gerät werden dann M-Kanäle ab dem am Gerät eingestellten Basiskanal N zugeordnet, also von N bis N+M-1. M darf deshalb maximal 16 betragen. Ist M gleich 0, so wird das Gerät angewiesen, alle seine Stimmen den Kanälen ab dem Basiskanal aufwärts zuzuordnen. Jedem Kanal muß immer genau eine Stimme zugeordnet werden. Diese Definitionen werden jedoch nur noch von wenigen älteren Geräten verwendet.
Der Mono-off-Befehl (126) schaltet den Poly-Mode aus bzw. den Poly-Mode ein (siehe 5.1.4). In 0vvvvvvv steht hierbei immer eine 0.
Merke: Alle Befehle von All Note-off (123) bis Mono on (127) wirken auch als All Note-off-Befehle. Sie schalten alle Töne, die auf den vom Basiskanal kontrollierten Stimmen

Tilo's Homepage-Website

liegen, ab. Sie sollten deshalb niemals nur als Note-off-Befehle mißbraucht werden, sondern immer nur zu ihrem speziellen Zweck gesendet werden. Deshalb sollten alle Empfänger, die keine Noten halten können, wie z.B. Drum-Machines, die Befehle 123-127 nicht als All Note-off interpretieren, da sowieso definitionsgemäß jedem Note-on-Befehl ein Note-off-Befehl folgen muß.

1100: Program Change (Programm-Änderung)

Syntax:	1100 nnnn	0ppp pppp
	Cn	Programm-Nummer

Beschreibung: Mit diesem Befehl kann das an einem Gerät bzw. für einen Kanal eingestellte Programm umgeschaltet werden.

Parameter:	0ppp pppp	Programm-Nummer (0-127)

Anmerkungen:
Mit diesem Befehl können die einzelnen Sound-Programme eines Gerätes umgeschaltet werden. Hierbei kann es sich sowohl um den Klang eines Synthesizers als auch die Parametereinstellungen an einem Effektgerät oder an einem Mixer handeln. Die jeweils zusammen verwendeten Einstellungen der einzelnen Geräte im System sollten deshalb vom User auf gleiche Programmnummern gelegt werden.

1101: After Touch (Gesamtdruck auf alle Tasten)

Syntax:	1101 nnnn	0vvv vvvv
	Dn	Druck-Wert

Beschreibung: Gibt eine Änderung des mechanischen Drucks auf alle Keyboard-Tasten an.

Parameter: 0vvv vvvv Druck-Wert (0-127). Bezeichnet den mechanischen Druck, der nach dem Anschlag auf die gesamte Klaviatur gegeben wird.

Anmerkungen:
Dieser Befehl wird bei jeder registrierten Änderung des Tastendrucks um ein Intervall gesendet. Deshalb nehmen diese Informationen normalerweise sehr viel Übertragungszeit auf dem MIDI-Bus und Speicherplatz im Sequencer in Anspruch. In Systemen, wo der After Touch nicht benötigt wird, sollte er ignoriert bzw. gar nicht erst gesendet werden.

1110: Pitch Wheel (Tonhöhen-Regler)

Syntax:	1110 nnnn	0111 1111	0mmm mmmm
	En	Pitch LSB	Pitch MSB

Beschreibung: Dient zur kontinuierlichen Änderung der Tonhöhe, meist mit Hilfe des Pitch-Benders an einem Keyboard.

Taken from Amiga-Manuals-Website

Parameter:	0111 1111	LSB des neuen Modulationswertes
	0mmm mmmm	MSB des neuen Modulationswertes

Anmerkungen:
Die Empfindlichkeit des Pitch-Benders wird am Empfangsgerät eingestellt. Die Mittelstellung ist auf den Wert hex 2000 festgelegt, der wie folgt übertragen werden kann: Ein (Statusbyte mit Kanalnummer) - 00 (LSB) - 20 (MSB).
Dieser Befehl wird bei jeder registrierten Änderung der Stellung des Pitch-Benders um ein Intervall gesendet. Deshalb nehmen diese Informationen normalerweise sehr viel Übertragungszeit auf dem MIDI-Bus und Speicherplatz im Sequencer in Anspruch.

1111: System Exclusive, System Common und System Realtime-Befehle

1111 0000: System Exclusive-Befehl

Syntax:	1111 0000	0iii iiii	[0ddd dddd]*	1111 0111
	F0	Hersteller-ID	Datenbytes	EOX (s.u.)

Beschreibung: Dient zur Übertragung von Hersteller-spezifischen Informationen.

Parameter:	0iii iiii	Hersteller-Identifikationsnummer. Muß bei der IMA (International MIDI Association) beantragt werden.
	[0ddd dddd]*	beliebig viele Datenbytes
	1111 0111	End of Exclusive (Statusbyte, siehe dort)

Anmerkungen:
Dieser Befehl dient zur Übertragung von Hersteller-spezifischen Informationen, wie z.B. die Übertragung von Wellenformen an Sampler u. Ä. Dem Statusbyte F0 folgt die Hersteller-Identifikationsnummer 0iii iiii, die leider nur 128 verschiedene IDs zuläßt. Darauf schließt sich eine beliebige Anzahl von Datenbytes an, deren internes Format durch die Art der Information bestimmt wird. Als normale Datenbytes müssen sie alle eine 0 im höchstwertigen Bit haben. Beendet wird die Sequenz durch das Statusbyte 11110111 (hex F7). Ein Empfänger, der einen System Exclusive Befehl nicht identifizieren kann, sollte die enthaltenen Bytes solange überlesen, bis EOX oder ein anderes Statusbyte empfangen wird. (Vorsicht: auch hier können Echtzeit-Befehle eingefügt sein!)

; System Common-Befehle (1111 0sss)

1111 0001 : Quarter Frame

Syntax :	1111 0001	0ttt dddd
	F1	ttt = Message Type dddd = Data

Taken from Manuals-Website

Die Quarter Frame Message dient zur Synchronisation von verschiedenen MIDI-Instrumenten im Studio und auf der Bühne und ist Bestandteil des sogenannten MTC (MIDI Time Codes). Diese Message wird nur während des Wiedergabe- und Aufnahmebetriebs übertragen. Sie beinhaltet eine System-Clock (Taktgeber) sowie die Time-Code-Zeit. Quarter Frame Messages werden immer in Blöcken zu je acht Messages übertragen. Nach dem Statusbyte (F1H) folgt das Byte, welches Message-Typ (siehe unten) sowie 4 Datenbit überträgt. Der Message-Typ ist unterteilt in Stunde und Time-Code-Type, Minute, Sekunde und Frame. Es sind zwei Quarter Frame Messages notwendig, um ein komplettes Datenbyte zu übermitteln, diese sind unterteilt in ein Low- und ein High-Nibble (untere und obere Bytehälfte).

Message-Type (ttt)		Data (High- und Low-Nibble)	
0 (0H)	Frame Low-Nibble		
1 (1H)	Frame High-Nibble	dddddddd	Frame 0-29
2 (2H)	Seconds Low-Nibble		
3 (3H)	Seconds High-Nibble	dddddddd	Sekunden 0-59
4 (4H)	Minutes Low-Nibble		
5 (5H)	Minutes High-Nibble	dddddddd	Minuten 0-59
6 (6H)	Hour Low-Nibble		
7 (7H)	Hour High-Nibble	dddddddd = xyyx xxxx	

yy = Time Code Type
0 : 24 Frames/s
2 : 25 Frames/s
4 : 30 Frames/s (NTSC)*
6 : 30 Frames/s (NTSC)**

z zzzz : Stunden 0-23

* Drop Frame, hier werden nach einem bestimmten Muster Frames ausgelassen, um effektiv eine Rate von 29,97 Frames/s zu erreichen
** Non Drop Frame, das Obige gilt hier nicht

[Quelle : KEYBOARDS 7/90,S. 117 ff., Musik-Media-Verlag]

1111 0010: Song Position Pointer

Syntax:	1111 0010	0111 1111	0mmm mmmm
	F2	LSB	MSB

Beschreibung: Dient zur Übertragung eines Zählerwertes, der die momentan gespielte Stelle in einem Song angibt. (Abstand vom Beginn des Song in 16tel Noten.)

Parameter:	0111 1111	LSB des Zählerwertes
	0mmm mmmm	MSB des Zählerwertes

Anmerkungen:
Dieser Befehl überträgt einen Zeiger auf die momentan gespielte Stelle in einem Song. Dies ist der Wert eines internen Zählers im Sequencer, der bei Betätigung des Start-

Knopfs (oder durch den MIDI-Startbefehl, s.u.) auf 0 gestellt und dann alle 6 MIDI-Clock-Intervalle um 1 inkrementiert wird. Er gibt also die Anzahl der 16tel Noten an, die seit dem Start des Songs verstrichen sind. Die Einstellung kann deshalb bis auf eine 16tel Note genau erfolgen.
Dieser Befehl wird im allgemeinen dazu benutzt, einen Sequencer extern auf eine bestimmte Position im gespeicherten Song zu richten. So könnte z.B. der eine Sequencer den einer Drum-Machine justieren, um beim Einspielen oder Abhören von bestimmten Passagen nicht immer wieder beim Beginn des Songs anfangen zu müssen. Der Befehl kann auch in Signale für eine Bandmaschine umgewandelt werden, die dann bei der Wiedergabe das System synchronisieren kann.

1111 0011: Song-Select-Befehl

Syntax:	1111 0011	0sss ssss
	F3	Song-Nummer

Beschreibung: Dient zur Anwahl eines im Sequencer gespeicherten Songs.

Parameter:	0sss ssss	Song-Nummer (0-127)

Anmerkungen:
Dieser Befehl bestimmt die Nummer des Songs, der nach dem Start-Befehl (s.u.) abgespielt wird.

1111 0100: nicht definiert

1111 0101: nicht definiert

1111 0110: Tune Request

Syntax:	1111 0110
	F6

Beschreibung: Veranlaßt alle angeschlossenen Geräte zur internen Durchstimmung.

Parameter: keine Parameter

Anmerkungen:
Dieser Befehl aktiviert die bei manchen Geräten vorhandene interne Durchstimm-Funktion (Auto-Tune). Diese stimmt das Gerät jedoch nur in sich selbst ab, die Stimmung mit anderen Geräten muß über den Tune-Regler erfolgen.

Taken from Amiga-Manuals-Website

1111 0111: EOX (End of Exclusive)

Syntax: 1111 0111
F7

Beschreibung: Beendet eine System-Exclusive-Datensequenz.

Parameter: keine Parameter

Anmerkungen:
siehe System Exclusive-Befehl

; System Realtime-Befehle (1111 1ttt)

1111 1000: Timing Clock

Syntax: 1111 1000
F8

Beschreibung: Dient zur Echtzeit-Synchronisation des Systems.

Parameter: keine Parameter

Anmerkungen:
Dieser Befehl synchronisiert alle am MIDI-Bus angeschlossenen Geräte, die einen eigenen Sequencer enthalten, wie z.B. viele Drum-Machines. Er dient lediglich als Taktsignal, aus dem die angeschlossenen Sequencer das Tempo entnehmen können, das am Master-Sequencer (Haupt-Steuerungsgerät) eingestellt ist. Der Timing-Clock-Befehl wird 24mal pro Viertelnote bzw. 6mal pro 16tel Note gesendet, und zwar unabhängig davon, ob sich der Master im Start- oder im Stopzustand befindet. Dadurch ist gewährleistet, daß bereits beim Start alle beteiligten Geräte im richtigen Tempo spielen.

1111 1001: nicht definiert

1111 1010: Start

Syntax: 1111 1010
FA

Beschreibung: Startet einen angeschlossenen Sequencer.

Parameter: keine Parameter

Anmerkungen:
Dieser Befehl dient zum Starten eines jeden am MIDI-Bus angeschlossenen Sequencers. Dieses Byte wird augenblicklich mit dem Drücken der Play-Taste am Master gesendet.

Taken from Amiga-Manuals-Website

1111 1011: Continue (Wiederfortsetzen)

Syntax: 1111 1011
FB

Beschreibung: Veranlaßt einen mit dem Stopp-Befehl angehaltenen Sequencer, die Wiedergabe fortzusetzen.

Parameter: keine Parameter

Anmerkungen:
Dieser Befehl dient zur Fortsetzung der Wiedergabe bei einem mit dem Stopp-Befehl angehaltenen Sequencer. Dieses Byte wird augenblicklich mit dem Drücken der Continue-Taste am Master gesendet.

1111 1100: Stop

Syntax: 1111 1100
FC

Beschreibung: Stoppt einen angeschlossenen Sequencer.

Parameter: keine Parameter

Anmerkungen:
Dieser Befehl dient zum Stoppen eines jeden am MIDI-Bus angeschlossenen Sequencers. Dieses Byte wird augenblicklich mit dem Drücken der Stopp-Taste am Master gesendet.

1111 1101: nicht definiert

1111 1110: Active Sensing

Syntax: 1111 1110
FE

Beschreibung: Dient zur Erkennung, ob am System ein Master aktiv ist.

Parameter: keine Parameter

Anmerkungen:
Dieser Befehl unterrichtet alle am System angeschlossenen Geräte von der »Anwesenheit« eines Masters am Bus. Er wird nur optional benutzt, kann also vom Master gesendet oder auch nicht gesendet, von einem Empfänger registriert oder ignoriert werden. Sofern am Bus keine anderen Aktivitäten vorgehen, wird die Active Sensing-Meldung alle 300 ms gesendet. Ein Empfänger erwartet diese Meldung normalerweise nicht. Wenn er sie aber

Taken from Amiga-Manuals-Website

MODEL JX-8P MIDI Implementation Chart

	Function.....	Transmitted	Recognized	Remarks
Basic	Default	1 – 16	1 – 16	memorized
Channel	Changed	1 – 16	1 – 16	
	Default	Mode 1, 3	Mode 1, 3	memorized
Mode	Messages	POLY, OMNI ON/OFF IO	POLY, OMNI ON/OFF	
	Altered	**********		MONO ignored
Note		36 – 96	0 – 127	
Number	True voice	**********	21 – 108	
Velocity	Note ON	O	*	v = 1–127
	Note OFF	X	X	
After	Key´s	X	X	
Touch	Ch´s	*	*	
Pitch Bender		*	*	
	1	*	*	Modulation
	5	*	*	Portamento Time
	7	X	*	Volume
Control	64	*	*	Hold
	65	*	*	Portamento Switch
Change				
Prog		* 0–127	* 0 – 127	
Change	True #	**********	0 – 127	
System Exclusive		*	*	
System	Song Pos	X	X	
	Song Sel	X	X	
Common	Tune	X	X	
System	Clock	X	X	
Real Time	Commands	X	X	
Aux	Local ON, OFF	X	O	Default ON
	All Notes OFF	O (123)	O (123–127)	
Messages	Active Sense	*	*	
	Reset	X	X	
Notes		*: Can be set O or X manually, and memorized		

Mode 1: OMNI ON, POLY **Mode 2**: OMNI ON, MONO O: Yes
Mode 3: OMNI OFF, POLY **Mode 4**: OMNI OFF, MONO X: No

Taken from Roland-Manuals.Website

einmal empfangen hat, und sie für über 300 ms ausbleibt, schaltet er seine Stimmen ab und geht in die normale Arbeitsroutine über.

1111 1111: System Reset (alles zurücksetzen)

Syntax: 1111 1111
FF

Beschreibung: Dient zur Erkennung, ob am System ein Master aktiv ist.

Parameter: keine Parameter

Anmerkungen:
Dieser Befehl versetzt das System in den Zustand, den es direkt nach dem Einschalten hatte. Er sollte stets umsichtig angewandt werden, am besten nur manuell ausgelöst. Vor allem sollte sie nicht automatisch nach Einschalten eines Gerätes abgegeben werden.

In dem obigen Abschnitt wurden die am häufigsten verwendeten MIDI-Befehle und - Messages dargestellt. Ein vollständiger und umfassender Überblick der gesamten, zur Zeit gültigen, MIDI-Implementation würde den Rahmen dieses Buches sprengen. Wer jedoch weitergehende Informationen wünscht, kann sich an folgende Adresse in den USA wenden: The International MIDI Association, 5316 W. 57th St., Los Angeles, CA 90056, USA. (Ein internationales Rückkuvert wäre von Vorteil!)
Eine andere Möglichkeit bestände darin, sich an die führenden Software-Häuser (C-Lab-Hamburg; Steinberg - TSI GmbH Waldorf, u.s.w.) zu wenden.

5.1.8 Die MIDI-Implementation-Chart

(Aus: Richard Aicher, Das MIDI-Praxisbuch, 1988, Seiten 237–246, Signum Medien Verlag)

Vor dem Kauf jeglichen MIDI-Equipments sollte man sich bereits gründlichst über die midimäßigen Gegebenheiten des Gerätes informieren. Hierbei ist ein Blick in die sogenannte MIDI-Implementation des Gerätes unumgänglich. Was ist das, weshalb benötigt man sie und was kann man als Musiker daraus entnehmen?

Jeder Synthesizer, jede Drum-Machine, jedes Masterkeyboard, überhaupt jedes midikompatible Gerät, verfügt über besondere Eigenheiten. Und das ist gut so. Denn wäre ein Synthesizer wie der andere, so gäbe es keine Klangunterschiede, keine Klaviaturen mit verschiedenem Spielgefühl, keine unterschiedlichen Preislagen. Genau wie sich Synthesizer in ihren Parametern unterscheiden, unterscheiden sie sich auch in der Art und Weise, wie sie auf verschiedene MIDI-Informationen reagieren. Wir wissen, daß eine Klaviatur, die über keine Anschlagsdynamik verfügt, folglich auch keine die MIDI-Anschlagsdynamik betreffende Informationen abgeben kann. Auch kann keine Pitch-Bend-Information abgegeben werden, wenn kein Pitch-Bender an der Klaviatur vorhanden ist. Und ein Slave, dessen Tonhöhenbereich bei der MIDI-Note 33 beginnt, kann keinen Ton mit der MIDI-Note 6 spielen.

Laut MIDI-Spezifikation ist jeder Hersteller verpflichtet, dem Gerät eine MIDI-Implementationskarte beizufügen. Es handelt sich hierbei um eine Tabelle, aus der zu ersehen ist, welche MIDI-Informationen das Gerät empfangen, abgeben, beziehungsweise übermitteln kann. Das Aussehen dieser Implementationskarte ist genau festgelegt. Es ist egal, ob es sich bei dem Gerät um einen Synthesizer, einen Sequencer, eine Drum-Machine oder ein MIDI-Effektgerät handelt. Die MIDI-Implementationskarte ist auch von Musikern sehr leicht zu lesen und vermittelt einen schnellen Überblick über die midimäßigen Gegebenheiten des Gerätes.

Betrachten wir die MIDI-Implementationskarte des Roland-JX-8-P-Synthesizers. Jede MIDI-Implementationskarte wird von einer Überschrift mit der genauen Modellbezeichnung des beschriebenen Gerätes und einer Zusammenstellung der möglichen MIDI-Modes am Fußende der Tabelle eingerahmt. Ebenfalls am Fußende der Implementation findet man eine kleine Zeichenerklärung. Normalerweise werden nur die beiden Zeichen Kreis und Kreuz verwendet, der Kreis bedeutet Ja, das Kreuz Nein. Ein Kreis in der Implementationskarte zeigt also an, daß eine bestimmte MIDI-Funktion implementiert ist, das Kreuz bedeutet, diese Funktion fehlt.

Jede Implementationskarte ist in vier Spalten untergliedert. Die erste Spalte ist mit dem Begriff »Function« überschrieben. In dieser Spalte stehen die einzelnen Funktionen des betreffenden Gerätes. Der zweiten Spalte »transmitted« läßt sich jeweils entnehmen, ob die entsprechende Funktion vom Gerät in Form eines MIDI-Befehls abgegeben werden kann oder nicht. Spalte Nr. 3 ist mit »recognized« überschrieben. Sie besagt, welche der Funktionen das Gerät für sich auswerten kann und welche nicht. In Spalte 4 sind, falls erforderlich, nötige Bemerkungen zusammengefaßt. Auch am unteren Tabellenende ist in dem mit »Notes« bezeichneten Feld Platz für weitere nötige Bemerkungen reserviert.

Betrachten wir die einzelnen Zeilen der Spezifikation und ihre jeweilige Bedeutung. Die Reihenfolge der Zeilen ist genau festgelegt. Die Funktionen stehen also in jeder Implementationskarte an derselben Stelle.

Die erste Zeile der Tabelle ist immer für den Basic-Channel reserviert. In der »transmitted«-Spalte entnimmt man, auf welchen Kanälen, wenn überhaupt, das Instrument MIDI-Informationen abgeben kann. In der »received«-Spalte steht, auf welchen Kanälen, wenn überhaupt, MIDI-Informationen empfangen werden können. Die Basic-Channel-Zeile ist in der »Function«-Spalte durch die Begriffe »Default« und »Changed« in zwei weitere Zeilen untergliedert. Der Default-Wert ist immer der Wert, den das Gerät direkt nach dem Einschalten einnimmt. Hier steht 1–16. Dies bedeutet, der JX-8-P kann also nach dem Einschalten wahlweise auf jedem der 16 MIDI-Channels empfangen, beziehungsweise senden.

In der Spalte »Remarkes« finden wir in der Default-Zeile den Hinweis »Memorized«. Dies bedeutet, daß der JX-8-P den zuletzt vor dem Ausschalten aktuellen MIDI-Kanal speichert und genau diesen nach erfolgtem Wiedereinschalten automatisch als Default-Channel einstellt.

In der »Changed«-Zeile steht ebenfalls 1–16, und zwar sowohl in der »Transmitted«- als auch in der »Recognized«-Spalte. Dies bedeutet, daß der JX-8-P auf jedem der 16 MIDI-Channels senden und empfangen kann.

In der nächsten Spalte der MIDI-Implementationskarte finden wir unsere MIDI-Modes. Auch diese Zeile ist in drei Zwischenzeilen unterteilt, deren erste mit »Default« überschrieben ist. Sowohl in der Transmitted- als auch in der Recognized-Spalte der Default-Zeile der JX-8-P-Implementationskarte ist »Mode 1,3« eingetragen. Dies bedeutet wieder, daß sich der JX-8-P die in den MIDI-Funktionen gewählte Mode-Einstellung speichert. Der vor dem Ausschalten aktuelle Mode erscheint auch nach dem Wiedereinschalten.

Die Eintragung »Mode 1,3« in der Transmitted-Spalte bedeutet, daß bei jeder Änderung der Mode-Einstellung am JX-8-P, an seinem MIDI-Out die entsprechende Mode-Information an die angeschlossenen Slaves ausgegeben wird.

Mode 1,3 in der Recognized-Spalte besagt, daß der JX-8-P über seinen MIDI-Input eintreffende Mode-Informationen registriert und auswertet, das heißt, sich automatisch in den befohlenen Mode begibt.

In der »Transmitted«-Spalte der mit »Altered« bezeichneten Zeile erkennt man eine ganze Reihe Sternchen. Sie bedeuten, daß die Default-Modes nicht geändert werden können. In der Spalte »Remarks« findet man die Bemerkung »Mono ignored«. Und dies ist auch die Lösung dieses kleinen Geheimnisses. Der JX-8-P ist ein polyphoner Synthesizer, er kann sich deshalb nur im Poly-Mode befinden, muß also jede Aufforderung, in den Mono-Mode zu schalten, ignorieren. Die nächste Zeile ist mit »Note-Number« bezeichnet. Hier finden wir Auskunft über den Tonumfang des Gerätes. In der »Transmitted«-Spalte steht, welchen Tonbereich das Gerät via MIDI senden, in der »Recognized«-Spalte, welchen Bereich es empfangen kann. Beide Angaben erfolgen in Form von MIDI-Notes. Der JX-8-P kann also die MIDI-Notes 36–96 senden, dies entspricht dem Tonhöhenbereich seiner Klaviatur. Empfangen kann er jedoch den gesamten MIDI-Note-Bereich von 0–127. Diese Tonhöheninformationen werden registriert und ausgewertet. Daß der JX-8-P jedoch den gesamten MIDI-Note-Bereich von 0–127 empfangen kann, heißt jedoch nicht, daß die empfangenen MIDI-Notes auch tatsächlich in der entsprechenden Tonhöhe wiedergegeben werden können. Diesen tatsächlich wiedergegebenen Tonhöhenbereich findet man in der Zeile »tru voice«. Der JX-8-P kann also die MIDI-Notes 21–108 tatsächlich wiedergeben. Trifft er eine kleinere bzw. größere MIDI-Note-Number an, so wird diese registriert, nicht aber in der Originaltonhöhe, sondern konstruktionsbedingt eine Oktave höher oder tiefer wiedergegeben.

Die nächste Spalte heißt »Velocity«. Hier wird es interessant. Wir sehen, daß die Velocity-Zeile in zwei Unterzeilen, eine für den Note-On-Befehl und eine für den Note-Off-Befehl, untergliedert ist.

In der Spalte »Remarks« steht, daß der JX-8-P die Velocity-Werte 1–127 annehmen kann. Betrachten wir die Note-On-Velocity. Der Kreis in der Transmitted-Spalte besagt: Diese Funktion ist implementiert. Der Stern in der Recognized-Spalte bedeutet, daß er JX-8-P die mit der Note-on-Information mitgelieferte Velocity wahlweise empfangen oder ablehnen kann. In der Note-off-Spalte der Velocity-Zeile stehen zwei Kreuze. Dies bedeutet, der JX-8-P ist nicht in der Lage, den abgegebenen Note-off-Informationen eine Velocity-Information mitzugeben.

Nun sollte die Interpretation der After-Touch-Zeile eigentlich ein leichtes sein. Wir haben es hier wieder mit zwei Zeilen zu tun. Die erste ist mit »Keys«, die zweite mit »Channels« überschrieben. Offensichtlich ist der JX-8-P, dies zeigen die zwei Kreuze in den entsprechenden Spalten, nicht in der Lage, die Tasten-Aftertouch-Funktion zu senden oder zu empfangen. Hingegen kann die Kanal-Aftertouch-Funktion sowohl empfangen als auch abgegeben werden.

Die beiden Sternchen besagen wieder, daß beide Informationen wahlweise an- oder abgeschaltet werden können. Das gleiche gilt auch für den Pitchbender, wir finden die entsprechendenden Informationen in der folgenden Spalte. Der JX-8-P kann Pitchbend-Informationen wahlweise empfangen/senden oder nicht.

Sehr wichtig ist auch die nächste Spalte »Control Change«. Für Control-Change-Informationen, die das Gerät abgeben, beziehungsweise auswerten kann, ist eine Zeile reserviert. In der Spalte Remarks steht dann jeweils, um welche Art von Control-Change es sich handelt, in der Function-Zeile die entsprechende Control-Nummer. Der JX-8-P registriert also Modulation, Portamento-Time, Volume, Hold und Portamento-Switch. Bis auf die Volume-Information kann er die Control-Informationen auch über seinen MIDI-Out zur Steuerung externer Slaves abgeben. Würde man den JX-8-P etwa von einem KX-76-Masterkeyboard spielen, und einem der vier belegbaren Schieberegler des KX-76 die Controllernummer 7 zuordnen, könnte man mit diesem Schieberegler direkt vom Masterkeyboard aus die Lautstärke des JX-8-P steuern.

Sehr wichtig, vor allem für das Funktionieren größerer MIDI-Systeme, ist die folgende Zeile »Program Change«. Die Inkompatibilität der Geräte bezüglich Anzahl vorhandener Speicherplätze und der Organisation des Speichers ist schon bekannt. In dieser Zeile findet man, welche Program-Change-Informationen das Gerät abgeben, beziehungsweise empfangen und tatsächlich auswerten kann. Am problemlosesten funktioniert das System immer, wenn alle beteiligten Instrumente über dieselbe Anzahl von Preset-Speicherplätzen verfügen. Sind insgesamt 128 Preset-Speicherplätze vorhanden, können alle Möglichkeiten des MIDI-Systems tatsächlich voll genutzt werden. Der Implementation des JX-8-P entnimmt man, das alle 128 Programme via MIDI angewählt werden können. Außerdem kann er alle 128 Programm-Changes über MIDI ausgeben. Wählt man ein Programm mit den Funktionstasten des JX-8-P an, wird der Wert verschlüsselt via MIDI ausgegeben und die Slaves ebenfalls in das entsprechende Programm geschaltet. Der Eintrag in der Zeile »True« ist interessant für Instrumente mit weniger als 128 Programmen. Er besagt, welchem Bereich des angegebenen Program-Change-Bereiches tatsächlich eigene Presets zugeordnet sind, wie viele Preset-Plätze das Gerät also tatsächlich besitzt. Prinzipiell können Instrumente mit weniger als 128 Presets über MIDI eintreffenden Programmwechselbefehle mit Nummern, die ihre Preset-Zahl übersteigen, entweder ignorieren oder ausführen. Werden sie ausgeführt, so beginnt das Instrument ab seiner höchsten Presetnummer wieder mit der Nummer 1, das heißt, dieselben Programme werden wiederholt. Werden sie ignoriert, bleibt das zuletzt aktive Programm einfach stehen.

In der nächsten Zeile der Implementation, »System Excusive«, steht, ob entsprechende Informationen vom Gerät gesendet, beziehungsweise empfangen werden können oder nicht. Wie bekannt, sind diese Informationen wichtig in Verbindung mit Heimcomputern. Sie gestattet die Programmierung des Instrumentes über einen Computer beziehungsweise das Speichern und Laden der Sounds auf Diskette.

Die nächsten beiden Zeilen der Implementation sind den System-Common- und den System-Realtime-Informationen gewidmet. Diese Zeilen sind besonders interessant für Drum-Machines und Sequenzer beziehungsweise MIDI-Keyboards mit integrierten Sequenzern. Hier findet man 1., ob ein ein Sequenzer oder eine Drum-Machine Song-Position-Pointer-Informationen registriert oder abgibt, das heißt, ob das Instrument in der Lage ist, im Song automatisch eine bestimmte Stelle zu lokalisieren oder eine entsprechende Information über MIDI abzugeben, und 2., ob das Instrument gezielt bestimmte Songs erkennen kann, beziehungsweise eine entsprechende Information abgibt. Selbstverständlich kann der JX-8-P (er verfügt über keinen Sequenzer) diese Informationen weder erkennen noch abgeben, was die Kreuze an den entsprechenden Stellen zeigen.

Die Zeile »Tune« ist umgekehrt wieder für Keyboards interessant, nicht für Sequenzer oder Drum-Machines. Die beiden Kreuze in der Tune-Zeile zeigen, daß der JX-8-P nicht in der Lage ist, eine eintreffende Auto-Tune-Funktion zu verarbeiten, beziehungsweise eine solche abzugeben.

Selbstverständlich ist der JX-8-P wieder resistent gegen die MIDI-Information der beiden System-Realtime-Zeilen: Clock und Commands. Sie betreffen wieder Sequenzer und Drum-Machines. Sequenzer oder Drums sollten in diesen beiden Zeilen sowohl in der Received- als auch in der Transmitted-Spalte einen Kreis aufweisen. Dann lassen sie sich synchronisieren (Clock) und auch vom Master aus starten, stoppen und wiederstarten (Commands).

Wir sind nun bei der letzten wichtigen Zeile der MIDI-Implementation angelangt. Hier steht, wie sich das Instrument bezüglich der sogenannten »Aux-Messages« verhält. Hierzu zählen Local-on/off, All Notes Off, Active Sense, Reset. Der Informationscharakter dieser Befehle ist bestens bekannt. Für den JX-8-P ist der Local-on/off-Zeile zu entnehmen, daß diese Information zwar empfangen, nicht aber gesendet werden kann. »Default On« besagt, daß sich der JX-8-P direkt nach dem Einschalten im Local-on-Mode befindet, das heißt, Klaviatur und Klangerzeugungsteil miteinander verbunden sind.

Der JX-8-P-Synthesizer sendet wie alle Roland-Keyboards nach polyphonem Spiel jedesmal mit dem Loslassen der letzten am Keyboard gedrückten Taste einen All-Notes-Off-Befehl. Damit werden die Slaves informiert, daß nun überhaupt keine Taste mehr gedrückt ist. Yamaha-Keyboards machen dies nicht. Sie senden lediglich die separaten, den einzelnen gespielten Tönen zugeordneten Note-Off-Befehle. Wie man der zugehörigen Zeile der Implementation entnimmt, versteht der JX-8-P diese Information auch.

Der Stern in der Active-Sensing-Zeile besagt, daß diese Funktion optional über die MIDI-Funktion einstellbar ist, und zwar sowohl für den Empfang als auch das Senden. Und nun sind wir ganz am Ende: Die letzte Zeile der Implementation besagt mit ihren beiden Kreuzen, daß der JX-8-P über den MIDI-Kanal weder eine Reset-Information empfängt, noch eine solche absenden kann.

Mit der genauen Kenntnis der MIDI-Implementation eines Instruments kann man die wichtigsten Aussagen über das voraussichtliche Verhalten des Gerätes in einem bestimmten MIDI-System treffen, ohne das Gerät überhaupt gesehen, beziehungsweise getestet zu haben. Damit ist die MIDI-Implementation ein unabdingbarer Helfer bei der Wahl des optimalen Gerätes und sollte vor jedem Kauf genauestens studiert und ausgewertet werden.

5.2 Das MIDI-Device

Wie wohl schon im letzten Abschnitt klargeworden ist, ist das Interpretieren oder Erzeugen eines MIDI-Datenstromes keine ganz triviale Aufgabe. Es sind hier eine ganze Menge verschiedener Eigenheiten des MIDI-Codes zu berücksichtigen, wie z.B. die Unterscheidung zwischen Status- und Datenbytes oder »normalen« und Realtime-Statusmeldungen, die richtige Anzahl von Datenbytes und vieles mehr. Noch verzwickter wird es, wenn man die Multitasking-Umgebung im Amiga berücksichtigt: Zwei Programme, die gleichzeitig vom seriellen Port versuchen zu lesen, können natürlich niemals korrekte Daten erhalten. Sie würden sich stets wichtige Teile der eingetroffenen Botschaften gegenseitig wegnehmen, so daß beide nur verstümmelte oder zumindest unvollständige Informationen erhielten.

Dasselbe Problem stellt sich auch bei der Ausgabe von MIDI-Daten: Senden mehrere Teilnehmer unkontrolliert Botschaften auf den Bus, so werden sich die einzelnen Bytes mit großer Wahrscheinlichkeit überschneiden und dadurch unsinnige Datenfolgen ergeben. Man benötigt hier also einen Softwarebaustein, der die Verteilung und Verwaltung sowohl der ankommenden als auch der gesendeten Messages übernimmt. Dieser sollte einerseits in der Lage sein, aus den hereinkommenden Botschaften die gewünschten auszufiltern und für jeden Empfänger bereitzuhalten; andererseits muß er alle Daten, die von den beteiligten Programmen gesendet werden, zu einer gültigen Folge zusammenfassen.

5.2.1 MIDI als Device

Für eine solche Aufgabe bietet sich beim Amiga die Form des Device an (siehe Kap. 3). Das MIDI-Device, welches den Softwareteil dieses Pakets abrundet, stellt eine Lösung für die oben genannten Probleme zur Verfügung. Wie alle Devices, besitzt es eine eigene Task, die für die korrekte Abwicklung der Lese- und Schreibvorgänge sorgt. Die Verteilung der Botschaften auf die einzelnen Empfänger wird (naheliegenderweise) über Units vorgenommen. Ein Unit entspricht also einer der oben skizzierten Empfangsstellen, an denen jedes angekommene MIDI-Event so lange aufgehoben wird, bis das jeweilige Programm es abholt. Insgesamt stehen 32 Units zur Verfügung.

MIDIEvent-Typen

Auch kann einem Unit mitgeteilt werden, welche Sorten von Events man überhaupt empfangen möchte. So ist es z.B. möglich, ein Unit für den Empfang von Notendaten, eines für Realtime-, und ein weiteres für SysEx-Messages einzusetzen, damit die jeweils zuständigen Tasks vollkommen unabhängig voneinander arbeiten können. Die möglichen Event-Kombinationen unterteilt das MIDI-Device in derzeit 21 verschiedene Typen, unter denen sich neben eigenen Typ-Nummern für Continuous Control, Switches und Mode auch zwei Pseudo-Events befinden, die den erstmaligen Empfang bzw. das Abreißen des Active-Sensing-Signals anzeigen. Welche Typen ein Unit bereithalten soll, wird in einer Flag-Variablen in der Unit-Struktur festgelegt, deren Wert jederzeit verändert werden kann. Folgende Typ-Flags sind für das MIDI-Device definiert:

```
#define METF_CHANNEL            0x01ff
#define METF_NOTEOFF            (1<<0)
#define METF_NOTEON             (1<<1)
#define METF_POLYPRESS          (1<<2)
#define METF_CCTRL              (1<<3)
#define METF_PROG               (1<<4)
#define METF_AFTERTOUCH         (1<<5)
#define METF_PITCHBEND          (1<<6)
#define METF_SWITCH             (1<<7)
#define METF_MODE               (1<<8)

#define METF_SYSCOMMON          0xe00
#define METF_SONGPOS            (1<<9)
#define METF_SONGSEL            (1<<10)
#define METF_TUNEREQ            (1<<11)

#define METF_SYSREALTIME        0x3f000
#define METF_CLOCK              (1<<12)
#define METF_START              (1<<13)
#define METF_CONTINUE           (1<<14)
#define METF_STOP               (1<<15)
#define METF_ACTSENS            (1<<16)
#define METF_RESET              (1<<17)

#define METF_SYSEX              (1<<18) /* SysEx empfangen */

#define METF_ACTSENSOFF         (1<<19)
                /* Act. Sens. ausgeschaltet */
#define METF_ACTSENSON          (1<<20)
                /* Act. Sens. eingeschaltet */

#define METF_ILLEGAL            (1<<31) /* MIDI Data Error ! */

#define METF_0DATA              0x1bf800
                /* Events ohne Datenbytes */
#define METF_1DATA              0x4430 /* 1 Datenbyte  */
#define METF_2DATA              0x3FCF /* 2 Datenbytes */
#define METF_MSBLSB  (METF_CCTRL!METF_PITCHBEND!METF_SONGPOS)
                /* hochauflösende Werte */
```

Diese Flags werden beim Aufruf von *OpenDevice()* aus dem I/O-Request in die Unit-Struktur kopiert. Verändert man die Typ-Flags, so gilt diese Änderung nur für die später eintreffenden Events, alle bis dahin empfangenen, aber noch nicht ausgelesenen Events bleiben erhalten.

Realtime-Events

Diese Sorte von Events wird vom MIDI-Device in gewisser Weise bevorzugt behandelt. Senden sie z.B. eine große Anzahl von Noten-Events hintereinander und danach ein Realtime-Event, so wird dies sofort ausgegeben, und nicht erst nach der Abarbeitung der vorangegeangenen Events. Dies kommt vor allem der Genauigkeit bei zeitkritischen Signalen zugute, wie z.B Start, Continue oder Clock.

Textbuch-Manuals-Website

System Exclusive Messages

SysEx-Botschaften werden ebenfalls auf besondere Weise behandelt: Alle Datenbytes, die zwischen den Statusbytes $F0 (System Exclusive) und $F7 (End of Exclusive) eintreffen, werden zwischengespeichert und können mit einem eigens dafür vorgesehenen Befehl ausgelesen werden. Ein solcher Befehl existiert auch für das Senden von SysEx-Botschaften.

Active Sensing

Das MIDI-Device erlaubt zwar auch den direkten Empfang von Active-Sensing-Events, bietet aber eine noch wesentlich komfortablere Behandlung dieses Signals. Setzen Sie die *METF_ACTSENSON*- und *METF_ACTSENSOFF*-Flags in ihrem Unit, so werden Sie bei jedem Ein- und Ausschalten eines angeschlossen Gerätes eine entsprechende Nachricht in Form eines Pseudo-Events erhalten.

Hochauflösende Werte

Einige MIDI-Events, namentlich Continuous Control, Pitch-Bender und Song Position Pointer, übertragen hochauflösende Werte, für deren Darstellung 2 Byte notwendig sind. Da MIDI-Datenbytes jedoch erst in ein 14-Bit-Format umgewandelt werden müssen, stellt das Device diese Werte gesondert zur Verfügung.

Die Continuous-Control-Events nehmen hier wiederum eine Sonderstellung ein: Hier werden nicht unbedingt immer beide Bytes gesendet, sondern nur der gerade benötigte Teil des Wertes. Deshalb speichert das Device sämtliche empfangenen Werte und zeigt entsprechend die Kombination aus dem gerade empfangenen und den letzten Werten an.

5.2.2 Die Strukturen des MIDI-Device

Für das MIDI-Device existiert natürlich auch eine eigene Variation der I/O-Request-Struktur, die IOMIDI getauft wurde. Diese ist in C so definiert:

```
struct IOMIDI {
  struct IOStdRequest IOM_IOStdR;
  UBYTE IOM_UnitNum;
  UBYTE IOM_Pad;
  ULONG IOM_TypeFlags;
};
```

IOM_IOStdR:

IOStdRequest-Struktur. Die Elemente *io_Device* und *io_Unit* werden beim Aufruf von *OpenDevice()* initialisiert, der Rest vom Anwender-Programm. Vor allem beim Lesen sollte *mn_ReplyPort* auf einen gültigen Message-Port zeigen, um Signale über den Abschluß von I/O-Vorgängen zu erhalten.

IOM_UnitNum:

In diesem Element ist die Nummer des Units angegeben, das mit diesem Request angesprochen wird. Da nur 32 Units zur Verfügung stehen, muß die Nummer zwischen 0 und 31 liegen.

Die Unit-Nummer korrespondiert eng mit dem *io_Unit*-Element, da das Device aus Geschwindigkeitsgründen teils dieses und teils die Unit-Nummer verwendet. Möchte man ein Unit also mit einem anderen Request ansprechen, so muß man den Wert in *IOM_UnitNum* stets zuerst in diesen übertragen.

IOM_TypeFlags

Hier wird z.B. bei *OpenDevice()* angegeben, welche Events ein Unit empfangen soll und welche ignoriert werden. Bei Lesevorgängen werden hier die Typen der gelesenen Events vermerkt.

Mit jedem Aufruf von *OpenDevice()* wird also eine Unit-Struktur erzeugt, und ihre Adresse im *io_Unit*-Element des Request eingetragen. Diese Struktur sieht folgendermaßen aus:

```
struct MIDIDevUnit {
  struct Unit mdu_Unit;
  UBYTE mdu_Flags;
  UBYTE mdu_UnitNum;
  ULONG mdu_RcvFlags;
  struct list mdu_ReadReqList;
};
```

mdu_Unit	Unit-Struktur
mdu_Flags	interne Flags
mdu_UnitNum	Unit-Nummer
mdu_RcvFlags	Typ-Flags der zu empfangenden Events
mdu_ReadReqList	Liste der noch nicht beendeten Lese-Requests

Wurde das Device erfolgreich geöffnet, ist es für Senden und Empfang von Daten über das jeweilige Unit bereit. Wie aber soll das Device die erhaltenen Daten darstellen bzw. interpretieren? Da MIDI-Botschaften verschiedene Längen und Formate haben und daher recht unhandlich sind, wurde für die Übertragung wiederum eine Struktur definiert:

```
struct MIDIEvent {
  APTR me_NextEvent;
  UBYTE me_Status;
  UBYTE me_Channel;
  UBYTE me_Data1;
  UBYTE me_Data2;
  WORD me_MsbLsb;
  LONG me_SysExLen;
  struct timeval me_TimeStamp;
};
```

Translation Amiga-Manuals Wiese

me_NextEvent: Wenn mehrere Events gelesen wurden, enthält dieses Element einen Zeiger auf die nächste MIDI-Event-Struktur, sonst 0.

me_Status: Status-Byte des Events.

me_Channel: Falls das Event kanalbezogen ist, so steht hier beim Lesen die Kanalnummer (von 0–15). Beim Schreiben wird dieses Element mit *me_Channel* ODER-verknüpft, so daß man das effektive Status-Byte auf zwei Arten definieren kann: erstens mit dem Status-Byte in *me_Status* und einer 0 in *me_Channel*, oder mit 0 in den Bits 0 bis 3 in *me_Status* und der Kanalnummer in *me_Channel*.

me_Data1: Erstes Datenbyte des Events.

me_Data2: Zweites Datenbyte des Events.

me_MsbLsb: Falls das Event einen hochauflösenden Wert überträgt, so wird dieser hier in 14-Bit-Darstellung angegeben. Ist beim Senden die normale MIDI-Darstellung gewünscht (in *me_Data1* und *me_Data2*), so muß dieses Element –1 enthalten.

me_SysExLen: Wurde eine SysEx-Message empfangen, so wird dies in Form eines Events mitgeteilt, das in *me_Status* das Status-Byte *MS_SYSEX* und in *me_SysExLen* die Länge der Message in Bytes enthält.

me_TimeStamp: Dieses Element enthält eine TimeVal-Struktur, in der die Systemzeit zum Zeitpunkt des Eintreffens des Events angegeben ist.

5.2.3 Funktionen und Kommandos

System-Funktionen

Das MIDI-Device kann mit allen dafür vorgesehenen Systemfunktionen angesprochen werden. Zuerst muß ein Device, wie schon mehrfach erwähnt, vor der Verwendung durch *OpenDevice()* geöffnet werden. Üblicherweise wird man hierbei so vorgehen:

```
int unitnum, error;
struct MsgPort *myPort;
struct IOMIDI *myIOM;

if(!(myPort = CreatePort("MIDI-Port",0) {
   printf("Konnte Port nicht erstellen !");
   exit();
}
if(!(myIOM=(struct IOMIDI *) CreateExtIO(myPort,sizeof(struct IOMIDI)) )) {
   printf("Konnte IO nicht erstellen !");
   DeletePort(myPort);
   exit();
}

unitnum = -1; /* nächstes freies Unit */
myIOM->IOM_TypeFlags = ~0; /* alle Events empfangen */
```

```
error = OpenDevice("midi.device", unitnum, myiom, 0);
if(error) {
   printf("Konnte midi.device nicht öffnen");
   DeleteExtIO(myIOM);
   DeletePort(myPort);
   exit();
}
```

Die Routinen *CreatePort()* und *DeletePort()*, ebenso *CreateExtIO()* und *DeleteExtIO()* sind Teil von amiga.lib, die mit jedem guten Assembler oder Compiler mitgeliefert wird (auch auf Public Domain erhältlich). Informationen über diese und andere Funktionen der *amiga.lib* finden sich in [Addison-Wesley, Amiga ROM Kernel Manual: Exec].

Die Unit-Nummer kann entweder direkt angegeben (0–31), oder durch Angabe von –1 das nächste freie Unit gewählt werden. Wählt man ein bestimmtes Unit, so ist zu beachten, daß ein Unit nur einmal (!) geöffnet werden kann. Versucht man es ein zweitesmal zu öffnen, so wird *OpenDevice()* eine Fehlernummer in d0 zurückgeben.

Ab dem Öffnen des Device, mit dem ja auch immer ein neues Unit reserviert wird, werden alle hereinkommenden MIDI-Botschaften zwischengespeichert, die von einem der geöffneten Units benötigt werden. Aus diesem Grunde sollten alle verwendeten Units möglichst immer sofort ausgelesen werden, um einen hohen Speicherplatzverbrauch und die damit verbundene Segmentierung des Speichers zu vermeiden.

Der Gebrauch der Funktionen *CloseDevice()*, *DoIO()*, *SendIO()*, *WaitIO()* und *CheckIO()* unterscheidet sich in nichts von den anderen Devices.

Device-Funktionen

Außer den obligatorischen System-Einsprüngen *BeginIO()* und *AbortIO()* bietet das MIDI-Device noch zwei weitere Funktionen:

MEventType(): Diese Funktion liefert die Typ-Flag eines Events als 32-Bit-Wert. In A1 muß dabei ein Zeiger auf die zu prüfende MEvent-Struktur enthalten sein. Diese Funktion wird folgendermaßen angewendet:

```
struct MIDIEvent *me;
ULONG  type;

type = MEventType(me);
```

MEventTypeBit(): Diese Funktion liefert die Typ-Nummer eines Events (derzeit 0–20, oder 31 für illegal). In A1 muß dabei ein Zeiger auf die zu prüfende MEvent-Struktur enthalten sein. Diese Funktion wird folgendermaßen angewendet:

```
struct MIDIEvent *me;
int    typenum;

typenum = MEventTypeBit(me);
```

Die Kommandos des MIDI-Device unterteilen sich in die Standard- und die Device-spezifischen (NonStandard-)Kommandos. Die Standard-Kommandos haben folgende Bedeutung:

OCMD_RESET: Zur Zeit nicht implementiert.

CMD_CLEAR: Dieses Kommando löscht den gesamten Empfangspuffer eines Units.

CMD_STOP: Hält das Unit an. Ab diesem Zeitpunkt werden alle weiteren Requests, die an das Unit gerichtet werden, in den Wartezustand versetzt, bis das Kommando *CMD_START* gegeben wird.

CMD_START: Startet ein durch *CMD_STOP* angehaltenes Unit wieder.

CMD_FLUSH: Wenn dieses Kommando gegeben wird, werden alle am Unit anhängigen Requests abgebrochen und mit einer Fehlermeldung (*IOERR_ABORTED*) beantwortet.

Dies sind die Device-spezifischen Kommandos:

MDCMD_GETUNIT: Dieses Kommando reseviert ein weiteres Unit. Hierbei muß in *IOM_UnitNum* entweder die Unitnummer selbst oder –1 (= nächstes freies Unit) angegeben werden. Das Feld IOM_RcvFlags muß ebenfalls initialisiert sein. Durch die Ausführung wird sowohl die effektive Unit-Nummer als auch der Zeiger in *IO_UNIT* korrekt gesetzt.

MDCMD_FREEUNIT: Alle Units, die nicht mit *OpenDevice()*, sondern mit *MDCMD_GETUNIT* erzeugt wurden, müssen mit diesem Kommando wieder freigegeben werden. Auch hier müssen Unitnummer und -Zeiger korrekte Werte aufweisen!

MDCMD_GETUNITFLAGS: Dieses Kommando überträgt den Wert von *mdu_RcvFlags* aus dem Unit, welches der Request anspricht, in das *IOM_TypeFlags*-Feld des Requests.

MDCMD_SETUNITFLAGS: Dieses Kommando überträgt den Wert von *IOM_TypeFlags* aus dem Request in das Unit. Dadurch werden die Event-Typen, die durch *MDCMD_READMEVENT* gelesen werden können, neu bestimmt. Dies betrifft aber nur Events, die nach der Ausführung des Kommandos eintreffen, alle vorherigen, aber noch nicht ausgelesenen Events bleiben erhalten.

MDCMD_READMEVENT: Mit diesem Kommando kann von einem Unit gelesen werden. Dies geschieht, ähnlich wie bei allen Devices, indem *MIDIEvent*-Strukturen in einen vom Anwender bereitgestellten Zielbereich kopiert werden. In *io_Data* muß wie üblich ein Zeiger auf den Zielbereich stehen. Die Anzahl der übertragenen Events richtet sich nach dem Wert in *io_Length*, geteilt durch die Länge eines Events in Bytes.

Stehen an einem Unit genügend Events bereit um den Request zu »befriedigen«, so wird die Abarbeitung des Requests direkt durchgeführt (Quick-IO). In diesem Fall wird das *IOB_QUICK*-Bit in *io_Flags* unberührt gelassen. Sind keine bzw. nicht genug Events eingetroffen, so wird der Request in eine (gesonderte) Warteliste gestellt, und erst nach dem Eintreffen der angeforderten Events beantwortet. Dabei wird das *IOB_QUICK*-Bit auf 0 gesetzt. In *io_Actual* kann zu jedem Zeitpunkt bestimmt werden, wie viele Events bereits übertragen sind.

MDCMD_WRITEMEVENT: Dieses Kommando erlaubt das Senden von MIDI-Botschaften über ein Unit. In *io_Data* muß hierbei ein Zeiger auf einen zusammenhängenden Speicherbereich stehen, in welchem die zu sendenden MIDIEvent-Strukuren untergebracht sind. In *io_Data* muß (wie bei Read) die Anzahl der zu übertragenden Events in Bytes angegeben sein.

Die zum Senden bestimmten MIDI-Bytes werden stets sofort aus den in *io_Data* angegebenen Events bestimmt und in den Ausgabepuffer übergeben. Aus diesem Grund ist auch der Zeitpunkt der Beantwortung des Requests nicht mit der Beendigung des Sendevorgangs identisch! In der Version 1.0 hat der Ausgabepuffer eine Größe von 1 Kbyte.

Nur um es noch einmal zu betonen: Das Feld *mdu_TypeFlags*, das nur für die Auswahl der empfangenen Events benutzt wird, hat keinen Einfluß auf dieses Kommando. Über ein geöffnetes, gültiges Unit können alle Event-Typen gesendet werden.

MDCMD_READSYSEX: System Exclusive Messages werden, wie erwähnt, vom MIDI-Device gesondert behandelt. Da sie variable Längen aufweisen, kann hier keine fest definierte Struktur eingeführt werden, die sie vollständig beschreiben würde. Aus diesem Grund wurde hier ein Umweg eingeschlagen: Beim Empfang einer SysEx-Message wird zunächst überprüft, ob eines der geöffneten Units in *mdu_RcvFlags* sein Interesse daran bekundet. Ist dies der Fall, so wird jedem dieser Units ein MIDI-Event mit dem Status *MS_SYSEX* und der Anzahl der Datenbytes in *me_SysExLen* übermittelt. In *me_Data1* ist die Hersteller-ID und in *me_Data2* das zweite Datenbyte der Message enthalten.

Mit diesem Kommando können die *SysEx*-Messages dann in der Reihenfolge ihres Eintreffens ausgelesen werden. Hierzu muß in *io_Data* ein Zeiger auf den Speicherbereich stehen, in den die Datenbytes der Message kopiert werden sollen. In *io_Length* muß die Anzahl der zu übertragenden Bytes enthalten sein (ohne die Status-Bytes *MS_SYSEX* und *MS_EOX*).

Maximal werden natürlich nur so viele Bytes übertragen, wie die jeweilige Message aufweist. Wichtig: Wurde ein *MDCMD_READSYSEX*-Kommando ausgeführt, so ist diese *SysEx*-Message für das jeweilige Unit »gestorben«, d.h. bei der nächsten Ausführung dieses Kommandos wird bereits die nächste *SysEx*-Message gelesen. Nicht gelesene Bytes der vorhergehenden Message sind dann verloren (Abhilfe: Ein zweites Unit unterhalten, von dem die Message dann ganz ausgelesen werden kann!).

MDCMD_WRITESYSEX: Mit diesem Kommando können SysEx-Messages Übertragen werden. In *io_Data* muß der Zeiger auf den Speicherberich enthalten sein, dessen Inhalt gesendet werden soll. Dieser Bereich darf nur Datenbytes enthalten (Bit 7 = 0), die Status-Bytes *MS_SYSEX* und *MS_EOX* werden automatisch hinzugefügt. Während des Schreibvorganges kann in *io_Actual* die Anzahl der bereits übertragenen Bytes abgelesen werden. Der Request wird dann beantwortet, wenn alle Bytes gesendet worden sind.

Hinweis zu Version 1.0: In dieser Version beantwortet das Device *MDCMD_WRITESYSEX*-Requests innerhalb eines Interrupts. Sollte dies Probleme aufwerfen, so empfiehlt sich die Nutzung eines separaten Units mit eigenem Request und Message-Port.

MDCMD_ACTSENSON: Durch dieses Kommando veranlaßt das Device, ein Active-Sensing-Signal zu senden. Von diesem Zeitpunkt an wird alle 300 ms das Realtime-Statusbyte *MS_ACTSENS* auf den Bus gegeben.

MDCMD_ACTSENSOFF: Dieses Kommando schaltet das gesendete Active-Sensing-Signal ab. Die meisten MIDI-Geräte reagieren darauf mit der Stummschaltung aller internen Stimmen.

MDCMD_LOCALECHOON: Dieses Kommando bewirkt, daß von nun an alle durch den MIDI-IN-Anschluß hereinkommenden Events sofort wieder auf den Ausgang gegeben werden, um so eine Local-Echo-Funktion zu bieten.

MDCMD_LOCALECHOOFF: Schaltet die Local-Echo-Funktion wieder ab.

5.2.4 Lesen vom MIDI-Device

Synchroner IO

Der Empfang von MIDI-Botschaften über das Device geschieht mit Hilfe des *MDCMD_READMEVENT*-Kommandos. Die Nummer dieses Kommandos muß wie immer im *io_Command*-Feld des I/O-Requests enthalten sein. Des weiteren muß man noch den Zeiger auf den Zielbereich in *io_Data* und die Größe des Speicherbereiches, der mit MIDIEvents gefüllt werden soll, in *io_Length* angeben.

Das folgende Beispiel demonstriert, wie eine bestimmte Anzahl von MIDIEvents in einen Speicherbereich eingelesen wird:

```
#define EVENTS 10

struct IOMIDI *myIOM;

struct MIDIEvent MEvents[EVENTS];

myIOM->IOM_IOStdReq.io_Data = &MEvents[0];
myIOM->IOM_IOStdReq.io_Length = EVENTS * sizeof(struct MIDIEvent);
myIOM->IOM_IOStdReq.io_Command = MDCMD_READMEVENT;
DoIO(myIOM);
```

In diesem Beispiel wird der Task innerhalb von *DoIO()* so lange schlafengelegt, bis zehn Events am Unit eingetroffen und in den Vektor *MEvents[]* kopiert worden sind.

Während des Lesevorganges kann dabei im Element *io_Actual* stets der aktuelle Stand, also die Anzahl der bereits übertragenen Events mal der Länge einens Events, abgelesen werden.

Asynchroner IO

Eine Alternative zur Verwendung von *DoIO()* bietet die Routine *SendIO()*. Diese übergibt den Request an das Device, geht aber anschließend nicht in den Wartezustand bis zu seiner Beantwortung, sondern kehrt sofort zurück und erlaubt somit die Erledigung anderer Aufgaben, während der

Lesevorgang läuft. Die Beantwortung des Requests kann dann mit *WaitIO()* abgewartet oder mit *CheckIO()* überprüft werden:

```
struct Message io;

/* hier folgt zuerst dieselbe Sequenz wie oben, außer: */

SendIO(myIOM);

/* hier können andere Dinge erledigt werden */

/* prüfen, ob I/O-Request schon beantwortet : */

if(io = CheckIO(myIOM))
   printf("IO bereits beantwortet !");
else
   WaitIO(myIOM); /* auf Beantwortung warten */

GetMsg(myPort);

/* falls der Port von mehreren Requests verwendet wird: Remove(myIOM); */
```

Benutzt man den Port, an den der Request beantwortet werden soll, noch mit mehreren anderen Requests, so darf *GetMsg()* nicht zum Entfernen des Requests vom Port eingesetzt werden. Während des Zeitraums zwischen der Beantwortung des Requests und dem Aufruf von *GetMsg()* könnte nämlich ein weiterer Request eintreffen, der dann anstatt des gewünschten vom Port entfernt würde. Deshalb muß man in diesem Fall *Remove()* verwenden.

Direkter Device-Einsprung (BeginIO())

Statt den Umweg über die Systemfunktionen *DoIO()* oder *SendIO()* zu nehmen, kann man das Device auch direkt über *BeginIO()* ansprechen. Zur Kontrolle, ob ein Request direkt durchgeführt oder in die Warteschlange gestellt wurde, muß hierbei das *IOB_QUICK*-Bit im *io_Flags*-Element des Request gesetzt sein.

Die verschiedenen Device-Kommandos haben einen unterschiedlichen Effekt auf dieses Flag. Die folgenden Kommandos haben niemals einen Effekt darauf, werden also immer direkt ausgeführt:

- *CMD_CLEAR*
- *CMD_STOP*
- *CMD_START*
- *CMD_FLUSH*
- *MDCMD_GETUNITFLAGS*
- *MDCMD_SETUNITFLAGS*
- *MDCMD_ACTSENSOFF*
- *MDCMD_ACTSENSON*
- *MDCMD_LOCALECHOOFF*
- *MDCMD_LOCALECHOON*

Einige Kommandos löschen das *QUICK*-Flag oder auch nicht, je nachdem, ob der Request sofort ausgeführt werden konnte und das Unit nicht gestoppt war:

- *MDCMD_GETUNIT*
- *MDCMD_FREEUNIT*
- *MDCMD_READMEVENT*
- *MDCMD_WRITEMEVENT*
- *MDCMD_READSYSEX*

Das einzige Kommando, das nie als Quick-IO ausgeführt wird und das Flag somit immer löscht, ist *MDCMD_WRITESYSEX*.

Hier ein Beispiel für die Verwendung des *BeginIO()*-Einsprungs:

```
/* hier folgt zuerst dieselbe Sequenz wie oben, außer: */

myIOM->IOM_IOStdReq.io_Flags = IOF_QUICK;

BeginIO(myIOM);

if(myIOM->IOM_IOStdReq.io_Flags & IOF_QUICK == 0) {
   /* QuickIO konnte nicht durchgeführt werden */
   WaitIO(myIOM);
   GetMsg(myPort); /* Request wurde beantwortet -> entfernen */
}
```

5.2.5 Senden über das MIDI-Device

Um MIDI-Botschaften über das Device zu senden, bedient man sich ebenfalls einer *MIDIEvent*-Struktur, die man vor dem Schreibkommando mit den entsprechenden Werten versieht. Schreiboperationen werden mit Hilfe des *MDCMD_WRITEMEVENT*-Kommandos durchgeführt.

Das folgende Beispiel zeigt das Senden einer MIDI-Note:

```
struct IOMIDI *myIOM;

struct MIDIEvent MEvent;

myIOM->IOM_IOStdReq.io_Data = &MEvent;
myIOM->IOM_IOStdReq.io_Length = sizeof(struct MIDIEvent);
myIOM->IOM_IOStdReq.io_Command = MDCMD_WRITEMEVENT;

MEvent.me_Status = MS_NOTEON;
MEvent.me_Data1 = 60; /* mittleres C */
MEvent.me_Data2 = 127; /* volle Lautstärke */
DoIO(myIOM);

/* hier solange warten, bis die Note abgebrochen werden soll */
```

Taken from Amiga-Manuals-Website

```
MEvent.me_Data2 = 0; /* identisch mit NOTE OFF */
DoIO(myIOM);
```

5.2.6 Einschränkungen

Da das MIDI-Device wie alle Devices von einem Task abhängig ist, reagiert es zumindest in der Version 1.0 sehr empfindlich auf das Verbieten des Task-Switchings. Bereits das Allocieren größerer Speicherbereiche kann beim Betrieb des MIDI_Device zu hörbaren Verzögerungen führen. Dies wird in höheren Versionen wahrscheinlich dahingehend geändert werden, daß die Verarbeitung hereinkommender MIDI-Daten in den Serial-Interrupt selbst verlegt wird, so daß sie in jedem Fall sofort zur Verfügung stehen.

5.2.7 Installation

Das Amiga-Betriebssystem sucht nach externen Devices stets im logischen Laufwerk »DEVS:«, welches normalerweise dem Verzeichnis *»Devs«* der Bootdiskette zugeordnet ist. Aus diesem Grunde müssen Sie das *midi.device* in das Devs-Verzeichnis Ihrer Arbeitsdiskette kopieren mit der Sie booten, z.B. mit folgendem Befehl:

```
»COPY AEP:MidiDev/midi.device MeineDisk:Devs«
```

wobei MeineDisk der Name Ihrer Arbeitsdiskette sein muß. Um von der Ihnen verwendeten Programmiersprache aus das *midi.device* zu nutzen, müssen Sie noch einige Dateien hinzufügen:

```
ASSEMBLER
```

Um das midi.device über Assembler korrekt anzusprechen, kopieren Sie bitte die Dateien *midi.i* und *mididev.i*, die sich im Verzeichnis AEP:MidiDev/ASSEMBLER/INCLUDE befinden, in das INCLUDE-Verzeichnis Ihres Assemblers, z.B. mit den Befehlszeilen :

```
»CD AEP:MidiDev/ASSEMBLER/INCLUDE«
»COPY midi.i MeineDisk:INCLUDE«
»COPY mididev.i MeineDisk:INCLUDE«
```

Nun müssen Sie in Ihr Assembler-Programm, von welchem aus Sie das *midi.device* ansprechen wollen, noch folgende Zeilen zu Beginn einfügen:

```
»INCLUDE "midi.i"«
»INCLUDE "mididev.i"«
```

Um das *midi.device* in C anzusprechen kopieren Sie bitte die Dateien *midi.h* und *mididev.h*, die sich im Verzeichnis AEP:MidiDev/CC/INCLUDE befinden, in das INCLUDE-Verzeichnis Ihrer C-Arbeitsdiskette, z.B. mit den Befehlszeilen:

```
»CD AEP:MidiDev/CC/INCLUDE«
»COPY midi.h MeineDisk:INCLUDE«
»COPY mididev.h MeineDisk:INCLUDE«
```

Nun müssen Sie in Ihr C-Programm, von welchem aus Sie das *midi.device* ansprechen wollen, noch folgende Zeilen zu Beginn einfügen:

```
»#include <midi.h>«
»#include <mididev.h>«
```

5.2.8 Beispielprogramme

Das erste Beispielprogramm, das in diesem Zusammenhang präsentiert werden soll, trägt den Namen MIDIMon. Es ist in Assembler geschrieben und gibt sämtliche auf dem Bus ankommenden MIDI-Botschaften als Text in dasjenige CLI-Fenster aus, von welchem es gestartet wurde.

```
;*********************************************************************
;
;    midimon.asm (c) Nicolaus Wirsing 12.7.89
;
;    All Rights reserved.
;
;*********************************************************************


    ; Dieses Programm muß mit amiga.lib gelinkt werden!!!


    opt       d+,m+

    incdir     devpac:include/
    include    exec/exec_lib.i
    include    exec/memory.i
    include    exec/io.i
    include    libraries/dos.i
    include    libraries/dos_lib.i
    include    asmsupp.i
    include    mididev.i

    XDEF     _SysBase
    XDEF     _DOSBase
    XDEF     _stdout
```

Taken from Amiga-Manuals-Website

```
    ; Makros für amiga.lib-Funktionen

    XREF        _CreatePort

CREATEPORT    MACRO
    IFC       '\2',''
    pea       0
    ENDC

    IFNC      '\2',''
    move.l    '\2',-(a7)
    ENDC

    IFC       '\1',''
    pea       0
    ENDC

    IFNC      '\1',''
    pea       CREATEPORT_Name\@
    bra.s     CREATEPORT_End\@

CREATEPORT_Name\@:
    dc.b       \1
    EVEN

CREATEPORT_End\@:
    ENDC
    jsr       _CreatePort
    adda.w    #8,a7
    ENDM

    XREF        _DeletePort

DELETEPORT    MACRO
    move.l    \1,-(a7)
    jsr       _DeletePort
    addq.l    #4,a7
    ENDM


    XREF      _printf

PRINTF        MACRO
    IFNC      '\9',''
    move.l    \9,-(a7)
    ENDC

    IFNC      '\8',''
    move.l    \8,-(a7)
    ENDC
```

Taken from Amiga-Manuals-Website

```
    IFNC        '\7',''
    move.l      \7,-(a7)
    ENDC

    IFNC        '\6',''
    move.l      \6,-(a7)
    ENDC

    IFNC        '\5',''
    move.l      \5,-(a7)
    ENDC

    IFNC        '\4',''
    move.l      \4,-(a7)
    ENDC

    IFNC        '\3',''
    move.l      \3,-(a7)
    ENDC

    IFNC        '\2',''
    move.l      \2,-(a7)
    ENDC

    pea         APF_FSTRING\@
    jsr         _printf
    adda.w      #NARG*4,a7
    bra.s       APF_end\@

APF_FSTRING\@:
    dc.b        \1,0
    EVEN

APF_end\@:
    ENDM


NSTRING         MACRO
NSTRING\@
    dc.b        NSTRING_CNT\@
    dc.b        \1,0

NSTRING_End\@
NSTRING_CNT\@
    EQU         NSTRING_End\@-NSTRING\@
    ENDM


    movea.l     4.w,a6

    lea         dosname(PC),a1                  ; DOS-Library öffnen
    moveq       #0,d0
```

Taken from Amiga-Manuals-Website

```
    CALLSYS   OpenLibrary
    move.l    d0,_DOSBase

    LINKSYS   Output,d0                    ; und Output-Handle ermitteln
    move.l    d0,_stdout                   ; (wird von printf benötigt)

    PRINTF    <'MIDIMon by Nicolaus Wirsing',$0a,'All Rights Reserved.',$0a>

CREATEPORT                                 ; ReplyPort erstellen
    tst.l     d0
    beq       End1

    lea       MyIOM(PC),a1                 ; I/O-Request initialisieren
    move.b    #NT_MESSAGE,LN_TYPE(a1)
    move.w    #IOM_SIZE,MN_LENGTH(a1)
    move.l    d0,MN_REPLYPORT(a1)
                                           ; zu empfangende MIDIEvents bestimmen

    move.l    #(~0)^(METF_ACTSENS!METF_CLOCK),IOM_TYPEFLAGS(a1)
    moveq     #-1,d0                       ; nächstes freies Unit
    lea       mdname(PC),a0
    moveq     #0,d1

    CALLSYS   OpenDevice                   ; Device öffnen
    tst.l     d0
    bne       End2

    lea       SysExIOM(PC),a0              ; zweiten I/O-Request initialisieren
    lea       MyIOM(PC),a1
    moveq     #IOM_SIZE-1,d0

SEIOMlp
    move.b    (a1)+,(a0)+
    dbra      d0,SEIOMlp

    lea       MyIOM(PC),a1
    lea       MyME(PC),a0
    move.l    a0,IO_DATA(a1)
    move.l    #me_size,IO_LENGTH(a1)
    move.w    #MDCMD_READMEVENT,IO_COMMAND(a1)

    lea       SysExIOM(PC),a0
    move.w    #MDCMD_READSYSEX,IO_COMMAND(a0)

MidiMonLoop:
    lea       MyIOM(PC),a1                 ; MIDIEvent lesen
    bset      #IOB_QUICK,IO_FLAGS(a1)
    BEGINIO

    lea       MyIOM(PC),a1
    btst      #IOB_QUICK,IO_FLAGS(a1)      ; Event gelesen (QuickIO)?
    bne.s     QuickIO
```

Taken from Amiga-Manuals-Website

```
    movea.l   MN_REPLYPORT(a1),a2              ; nein: warten
    move.b    MP_SIGBIT(a2),d1
    moveq     #0,d0
    bset      d1,d0
    bset      #SIGBREAKB_CTRL_C,d0
    CALLSYS   Wait
    btst      #SIGBREAKB_CTRL_C,d0             ; CTRL-C gedrückt?
    bne       End1

    movea.l   a2,a0
    CALLSYS   GetMsg

QuickIO:
    lea       MyME(PC),a1
    movea.l   a1,a2
    LINKSYS   MEventType,MyIOM+IO_DEVICE(PC)   ; TypeFlag des Events

    move.l    d0,d4
    bpl.s     QuickIO_ok                       ; 0 ? illegal
    lea       NStr_Illegal(PC),a0
    bra.s     PNStr1

QuickIO_ok
    move.l    d1,d5                            ; d1 = TypeBitNr
    lea       NStringArray(PC),a0              ; Typ-String ermitteln
    subq.l    #1,d1
    bmi.s     PrintNStrOk
    moveq     #0,d0

PNSloop
    move.b    (a0),d0
    adda.l    d0,a0
    dbra      d1,PNSloop

PrintNStrOk
    addq.l    #1,a0

PNStr1
    PRINTF    <'%-15ls'>,a0                    ; und ausgeben

    move.l    #METF_CHANNEL,d0                 ; Channel ? -> ausgeben
    and.l     d4,d0
    beq.s     NoChannel
    moveq     #0,d0
    move.b    me_Channel(a2),d0
    PRINTF    <'Ch: %2ld  '>,d0

NoChannel
    move.l    #METF_1DATA,d0                   ; 1. DatenByte ? -> ausgeben
    and.l     d4,d0
    beq.s     2DATA
    moveq     #0,d0
```

Taken from Amiga-Manuals-Website

```
    move.b    me_Data1(a2),d0
    PRINTF    <'Data1: $%02lx     '>,d0

_2DATA
    move.l    #METF_2DATA,d0                ; 2. DatenByte ? -> ausgeben
    and.l     d4,d0
    beq.s     _MSBLSB
    moveq     #0,d0
    moveq     #0,d1
    move.b    me_Data1(a2),d0
    move.b    me_Data2(a2),d1
    PRINTF    <'Data1,2: $%02lx $%02lx  '>,d0,d1

_MSBLSB
    move.l    #METF_MSBLSB,d0               ; hochauflösender Wert?
    and.l     d4,d0
    beq.s     _SysEx
    moveq     #0,d0
    move.w    me_MsbLsb(a2),d0
    PRINTF    <'MsbLsb: $%04lx'>,d0

_SysEx
    move.l    #METF_SYSEX,d0                ; System Exclusive?
    and.l     d4,d0
    beq       PrintReturn
    moveq     #0,d0
    moveq     #0,d1
    move.b    me_Data1(a2),d0               ; ja: ID, 1. Byte
    move.b    me_Data2(a2),d1               ; und Länge ausgeben
    move.l    me_SysExLen(a2),d4
    PRINTF    <'ID: %3ld  1st Data: %3ld  Len: $%08lx'>,d0,d1,d4

    move.l    d4,d0                         ; SysEx einlesen
    moveq     #MEMF_PUBLIC,d1
    CALLSYS   AllocMem
    move.l    d0,a4
    lea       SysExIOM(PC),a1
    move.l    d0,IO_DATA(a1)
    move.l    d4,IO_LENGTH(a1)
    CALLSYS   DoIO

    PRINTF    <$0a>
    movea.l   a4,a3
    move.l    d4,d3
    subq.l    #1,d3
    moveq     #15,d5

SysExlp
    moveq     #0,d0
    move.b    (a3)+,d0
    PRINTF    <'%02lx '>,d0                 ; Bytes in hex ausgeben
    dbra      d5,SElpok
```

Tafemform Amiga-Manuals-Website

```
    moveq       #15,d5
    PRINTF      <$0a>

SElpok
    dbra        d3,SysExlp
    movea.l     a4,a1
    move.l      d4,d0
    CALLSYS     FreeMem

PrintReturn
    PRINTF      <$0a>
    bra         MidiMonLoop

End1
    lea         MyIOM(PC),a1                    ; aufräumen
    CALLSYS     CloseDevice

End2
    lea         MyIOM(PC),a1
    movea.l     MN_REPLYPORT(a1),a1
    DELETEPORT  a1

Endx
    movea.l     _DOSBase(PC),a1
    CALLSYS     CloseLibrary
    rts

MyIOM:
    ds.b        IOM_SIZE

SysExIOM:
    ds.b        IOM_SIZE

MyME:
    ds.b        me_size

_DOSBase:
    ds.l        1

_stdout:
    ds.l        1

mdname:
    MIDINAME

dosname:
    DOSNAME

NStringArray:
NStr_NoteOff    NSTRING     <'Note Off'>
NStr_NoteOn     NSTRING     <'Note On'>
NStr_PolyPress  NSTRING     <'Poly-Pressure'>
```

Taken from Amiga-Manuals-Website

```
NStr_CCtrl          NSTRING    <'Cont. Control'>
NStr_Prog           NSTRING    <'Prog. Change'>
NStr_AfterTouch     NSTRING    <'AfterTouch'>
NStr_PitchBend      NSTRING    <'PitchBend'>
NStr_Switch         NSTRING    <'Switch'>
NStr_Mode           NSTRING    <'Mode'>
NStr_SongPos        NSTRING    <'Song Pos.'>
NStr_SongSelect     NSTRING    <'Song Select'>
NStr_TuneReq        NSTRING    <'Tune Request'>
NStr_Clock          NSTRING    <'Clock'>
NStr_Start          NSTRING    <'Start'>
NStr_Continue       NSTRING    <'Continue'>
NStr_Stop           NSTRING    <'Stop'>
NStr_ActSens        NSTRING    <'Act. Sensing'>
NStr_Reset          NSTRING    <'Reset'>
NStr_SysEx          NSTRING    <'System Excl.'>
NStr_ActSensOff     NSTRING    <'ActSensOff'>
NStr_ActSensOn      NSTRING    <'ActSensOn'>

NStr_Illegal
    dc.b        "Illegal!!!",0

EVEN
```

Das zweite Beispielprogramm ist in C geschrieben und stellt eine kleine Simulation eines Synthesizer-Keyboards dar. Alle gespielten Töne werden sowohl als MIDI-Botschaft auf den Bus gegeben als auch mit Hilfe der SMUSPlayer-Library von den internen Tongeneratoren des Amiga gespielt.

Hierbei wird ein Fenster geöffnet, in welchem die folgenden Bedienungselemente untergebracht sind:

❑ Ein-/Ausschalter für ein Active-Sensing-Signal, welches über das *midi.device* alle 300 ms ausgegeben wird.

❑ Ein-/Ausschalter für die Local-Echo-Funktion; im eingeschalteten Zustand werden alle ankommenden MIDI-Botschaften über das *midi.device* wieder auf den MIDI-Ausgang gegeben.

❑ Eingabeelemente für MIDI-Empfangs- und Sendekanal. In den zwei Integer-Gadgets kann der Empfangs- (Rcv Channel) und Sendekanal (Send Channel) mit Werten von 1 bis 16 direkt eingegeben werden. Die Pfeil-Gadgets rechts daneben erhöhen/erniedrigen die Werte in 1er-Schritten. Über das nach unten gerichtete Rcv-Channel-Gadget kann auch AL angewählt werden (oder durch Eingabe von 0 im Integer-Gadget). In dieser Einstellung werden alle eintreffenden MIDI-Events, ungeachtet der Kanalnummer, bearbeitet (OMNI ON !).

❑ Tonausgabe-Status-Gadget; hierüber kann durch Anklicken angewählt werden, ob bei Betätigung der Klaviatur auf dem Bildschirm mit der Maus Noteninformationen nur über MIDI, nur an den Amiga oder an beide gesendet werden. Diese Einstellung zeigt auch Wirkung bei ankommenden MIDI-Events.

Tandy Amiga Manuals-Website

❑ Program-Change-Gadget; hierüber können die Sounds oder Programme eines angeschlossenen MIDI-Geräts in Werten von 1 bis 128 angewählt werden – auf gleichen MIDI-Sendekanal von Programm und Gerät achten!

❑ Pitchbend-Regler; zeigt MIDI-Pitchbend-Informationen, die über das *midi.device* empfangen werden optisch an und sendet solche auch bei Betätigung mit der Maus an angeschlossene MIDI-Instrumente.

❑ Modulations-Regler; wie oben, diesmal für MIDI-Modulations-Informationen. Wenn man ein angeschlossenes MIDI-Instrument spielt, werden die ankommenden MIDI-Noten auf der Bildschirmklaviatur optisch angezeigt!

Das Programm besteht aus drei Quelldateien, einem Include-File (MIDIDemo.h), dem Hauptprogramm (MIDIDemo.c) sowie den Routinen zur Verwaltung der Bildschirm-Klaviatur (Keyboard.c). Hier nun die drei Listings:

Das Programm MIDIDemo.h:

```
    /************************ MIDIDemo.h ************************/


         /*** Damit wir mit Kickstart 2.0 keine Überaschungen erleben ! ***/
struct TextAttr DefaultFont = {
                                (STRPTR)"topaz.font",TOPAZ_EIGHTY,FS_NORMAL,
                                FPF_ROMFONT
                              };

#define DFNT &DefaultFont
                                          /*** Die Pfeil-Gadget-Pfeile ***/

UWORD UpArrowData[] = {                    /*** Daten für den Up-Pfeil ***/
        0x0000,0x7FF0,0x7DF0,0x78F0,0x7070,0x6030,0x4010,0x7DF0,
        0x7DF0,0x7DF0,0x7DF0,0x7DF0,0x7FF0,0x0000,0xFFF8,0xFFF8,
        0xFDF8,0xF8F8,0xF078,0xE038,0xC018,0xFDF8,0xFDF8,0xFDF8,
        0xFDF8,0xFDF8,0xFFF8,0xFFF8
};

struct Image UpArrow = {                   /*** Up-Pfeil Image-Struktur ***/
                         0,0,13,14,2,UpArrowData,0x0003,0x0000,NULL
                       };

UWORD DownArrowData[] = {                 /*** Daten für den Down-Pfeil ***/
        0x0000,0x7FF0,0x7DF0,0x7DF0,0x7DF0,0x7DF0,0x7DF0,0x4010,
        0x6030,0x7070,0x78F0,0x7DF0,0x7FF0,0x0000,0xFFF8,0xFFF8,
        0xFDF8,0xFDF8,0xFDF8,0xFDF8,0xFDF8,0xC018,0xE038,0xF078,
        0xF8F8,0xFDF8,0xFFF8,0xFFF8
};
```

```
struct Image DownArrow =   {                /*** Down-Pfeil Image-Struktur ***/
                               0,0,13,14,2,DownArrowData,0x0003,0x0000,NULL
                           };

#define ASTEXTL 568                 /*** Text-Position-Left (X-Koordinate) ***/
#define ASTEXTT 40                  /***  Text-Position-Top (Y-Koordinate) ***/

#define FPEN     3                      /*** FrontPen für Window, Gadgets ***/
#define BPEN     1                      /***  BackPen für Window, Gadgets ***/

#define KEYBOARDL     90                    /*** Keyboard-Left (X-Coord) ***/
#define KEYBOARDT     49                    /***  Keyboard-Top (Y-Coord) ***/
#define KEYBOARDH     30                    /***      Keyboard-Height    ***/

#define MMF_NONE        0                   /***  MIDI-MessageType Flags ***/
#define MMF_KEYBOARD    1
#define MMF_MODWHEEL    2
#define MMF_PITCHWHEEL  3

#define NORMKEYCOLOR    0                           /*** Keyboard-Colors ***/
#define SLCTKEYCOLOR1   1
#define SLCTKEYCOLOR2   2

UBYTE   StringBuffer[3][4] = { "1","1","1" };

#define RCHS   ((APTR)&(ChString[0]))        /*** Receive-Channel-String ***/
#define SCHS   ((APTR)&(ChString[1]))         /***   Send-Channel-String  ***/
#define PCHS   ((APTR)&(ChString[2]))         /***  Program-Change-String ***/

                                   /*** Strings für die Integer-Gadgets ***/
static struct StringInfo ChString[3] = {
        {StringBuffer[0], 0, 0, 3, -1},
        {StringBuffer[1], 0, 0, 3, -1},
        {StringBuffer[2], 0, 0, 4, -1}
};

#define PPHPOT    0                 /***   ProportionalGadget HORIZPOT   ***/
#define PPVPOT    MAXPOT/2          /*** ProportionalPitchGadget VERTPOT ***/
#define PMVPOT    MAXPOT            /***  ProportionalModGadget VERTPOT  ***/
#define PPHBODY   0                 /***  ProportionalGadget HORIZBODY   ***/
#define PPVBODY   5                 /***   ProportionalGadget VERTBODY   ***/

#define PROPP  ((APTR)&(MD_Prop[0]))       /*** Pitch-Prop-Gadget-Adresse ***/
#define PROPM  ((APTR)&(MD_Prop[1]))       /***  Mod-Prop-Gadget-Adresse  ***/

                                           /*** Prop Mod & Pitch PropInfo ***/
struct PropInfo MD_Prop[2] = {
        { AUTOKNOB|FREEVERT, PPHPOT, PPVPOT, PPHBODY, PPVBODY }, /* Pitch */
        { AUTOKNOB|FREEVERT, PPHPOT, PMVPOT, PPHBODY, PPVBODY }  /*  Mod  */
    };
```

Taken from Amiga-Manuals-Website

```
                                    /*** Die Gadget-Umrandungs-Koordinaten ***/
WORD G0BCoords[] = {  0, 0, 37, 0, 37,13, 0,13, 0, 0 };
WORD G1BCoords[] = {  0,-1, 24,-1, 24, 8,-1, 8,-1,-1 };
WORD EnaCoords[] = {  0, 0,139, 0,139,12, 0,12, 0, 0 };
WORD PChCoords[] = {  0, 0, 34, 0, 34, 9,       0, 9,   0, 0 };

struct Border GBorder[4] = {                  /*** Die Gadget-Umrandungen ***/
   { 0, 0, FPEN, BPEN, JAM2, 5, G0BCoords, NULL },
   { 0, 0, FPEN,    3, JAM2, 5, G1BCoords, NULL },
   { 0, 0, FPEN, BPEN, JAM2, 5, EnaCoords, NULL },
   {-2,-1, FPEN, BPEN, JAM2, 5, PChCoords, NULL }
};

#define OOB    ((APTR)&(GBorder[0]))   /*** ActSens & LocEch-Border/Rand ***/
#define STB    ((APTR)&(GBorder[1]))       /*** StringGadget-Border/Rand ***/
#define ENB    ((APTR)&(GBorder[2]))       /*** EnableGadget-Border/Rand ***/
#define PCB    ((APTR)&(GBorder[3]))       /*** Progr-Change-Border/Rand ***/


struct IntuiText MIDIWinText[7] = {                /*** Alle Gadget-Texte ***/
   {FPEN,   2,JAM2,   7,  3,DFNT,(UBYTE *)"OFF",NULL},
   {FPEN,   2,JAM2,   7,  3,DFNT,(UBYTE *)"ON ",NULL},
   {FPEN,BPEN,JAM1, -90,  1,DFNT,(UBYTE *)"Rcv Channel",NULL},
   {FPEN,BPEN,JAM1, -98,  1,DFNT,(UBYTE *)"Send Channel",NULL},
   {FPEN,   2,JAM2,   2,  3,DFNT,(UBYTE *)"Amiga & MIDI play",NULL},
   {FPEN,   2,JAM2,   2,  3,DFNT,(UBYTE *)" only Amiga play ",NULL},
   {FPEN,   2,JAM2,   2,  3,DFNT,(UBYTE *)" only MIDI play ",NULL}
};

#define OFTX   MIDIWinText[0]
#define ONTX   MIDIWinText[1]
#define RCTX   MIDIWinText[2]
#define SCTX   MIDIWinText[3]
#define ENTX   MIDIWinText[4]


#define IMG0   ((APTR)&MD_Image[0])                /*** Pitch-Prop Image ***/
#define IMG1   ((APTR)&MD_Image[1])                  /*** Mod-Prop Image ***/

struct  Image MD_Image[2];   /*** ImageStrukturen für die PropGadgets ***/

#define ASONID    0                  /***    ActiveSensOn GadgetID      ***/
#define LEONID    1                  /***    LocalEnableOn GadgetID     ***/
#define RCHSID    2                  /*** ReceiveChannelString GadgetID ***/
#define RCHUID    3                  /***   ReceiveChannelUp GadgetID   ***/
#define RCHDID    4                  /***  ReceiveChannelDown GadgetID  ***/
#define SCHSID    5                  /***   SendChannelString GadgetID  ***/
#define SCHUID    6                  /***    SendChannelUp GadgetID     ***/
#define SCHDID    7                  /***   SendChannelDown GadgetID    ***/
#define PITWID    8                  /***       Pitch GadgetID          ***/
#define MODWID    9                  /***        Mod GadgetID           ***/
#define ENPLID   10                  /***     EnablePlay GadgetID       ***/
```

Taken from Amiga-Manuals-Website

```
#define PGCHNGE  11                    /***     PGChange    GadgetID        ***/
#define PGCHDWN  12                    /***     PGChangeDown GadgetID       ***/
#define PGCHUP   13                    /***     PGChangeUp GadgetID         ***/

#define G(n)    (&(MD_Gadget[n]))     /*** Adresse eines Gadgets bestimmen ***/

                                                        /*** Die 14 Gadgets ***/
struct Gadget MD_Gadget[14] = {
/*** NEXT, LFT, TP, WDTH, H, FLAG, ACT, TYP, REND, 0, TXT, 0,SPEC, ID, 0 ***/

                                                      /*** ActiveSens Gadget ***/
   {G(1),128,14,37,13,GADGHCOMP,RELVERIFY,
                          BOOLGADGET,OOB,0,&OFTX,0,0,0,0},

                                                       /*** LocalEcho Gadget ***/
   {G(2),258,14,37,13,GADGHCOMP,RELVERIFY,
                          BOOLGADGET,OOB,0,&ONTX,0,0,1,0},

                                          /*** ReceiveChannelString Gadget ***/
   {G(3),391,16,24,8,NULL,RELVERIFY|STRINGCENTER|LONGINT,
                          STRGADGET,STB,0,&RCTX,0,RCHS,2,0},

                                                 /*** ReceiveChannelUp Gadget ***/
   {G(4),432,13,13,14,GADGIMAGE|GADGHCOMP,GADGIMMEDIATE|RELVERIFY,
                                BOOLGADGET,(APTR)&UpArrow,0,0,0,0,3,0},

                                                /*** ReceiveChannelDown Gadget ***/
   {G(5),418,13,13,14,GADGIMAGE|GADGHCOMP,GADGIMMEDIATE|RELVERIFY,
                                BOOLGADGET,(APTR)&DownArrow,0,0,0,0,4,0},

                                                 /*** SendChannelString Gadget ***/
   {G(6),548,16,24,8,NULL,RELVERIFY|STRINGCENTER|LONGINT,
                                STRGADGET,STB,0,&SCTX,0,SCHS,5,0},

                                                    /*** SendChannelUp Gadget ***/
   {G(7),589,13,13,14,GADGIMAGE|GADGHCOMP,GADGIMMEDIATE|RELVERIFY,
                                BOOLGADGET,(APTR)&UpArrow,0,0,0,0,6,0},

                                                  /*** SendChannelDown Gadget ***/
   {G(8),575,13,13,14,GADGIMAGE|GADGHCOMP,GADGIMMEDIATE|RELVERIFY,
                                BOOLGADGET,(APTR)&DownArrow,0,0,0,0,7,0},

                                                          /*** Pitch Gadget ***/
   {G(9),23,49,20,30,0,GADGIMMEDIATE|FOLLOWMOUSE|RELVERIFY,
                        PROPGADGET,IMG0,0,NULL,0,PROPP,8,0},

                                                            /*** Mod Gadget ***/
   {G(10),64,49,20,30,0,GADGIMMEDIATE|FOLLOWMOUSE|RELVERIFY,
                        PROPGADGET,IMG1,0,NULL,0,PROPM,9,0},

                                                       /*** Enable Play Gadget ***/
   {G(11),156,33,138,12,NULL,RELVERIFY,
                        BOOLGADGET,ENB,NULL,&ENTX,NULL,NULL,10,NULL},
```

Taken from Amiga-Manuals-Website

```
                                                    /*** PGChange Gadget ***/
   {G(12),419,36,33,14,NULL,RELVERIFY|STRINGCENTER|LONGINT,
                        STRGADGET,PCB,NULL,NULL,NULL,PCHS,11,NULL },

                                                  /*** PGChange Down Gadget ***/
   {G(13),455,33,13,14,GADGIMAGE,RELVERIFY|GADGIMMEDIATE,
                      BOOLGADGET,(APTR)&DownArrow,NULL,NULL,NULL,NULL,12,NULL},

                                                    /*** PGChange Up Gadget ***/
   {NULL,469,33,13,14,GADGIMAGE,RELVERIFY|GADGIMMEDIATE,
                        BOOLGADGET,(APTR)&UpArrow,NULL,NULL,NULL,NULL,13,NULL}

};

#define WLEFT     0             /***  Window-Koordinaten (Left X-Coord)   ***/
#define WTOP      12            /***  Window-Koordinaten (Top Y-Coord)    ***/
#define WWIDTH    640           /***  Window-Koordinaten (Width X-Coord)  ***/
#define WHEIGTH   86            /*** Window-Koordinaten (Height Y-Coord) ***/

                                                /*** Das Fenster von MIDIDemo ***/
struct NewWindow MIDIDemoWindow = {
        WLEFT,WTOP,
        WWIDTH,WHEIGTH,
        3,2,
        MOUSEBUTTONS|MOUSEMOVE|GADGETDOWN|GADGETUP|RAWKEY|CLOSEWINDOW,
        WINDOWDRAG|WINDOWDEPTH|WINDOWCLOSE|SMART_REFRESH|REPORTMOUSE|ACTIVATE,
        NULL,NULL,
        (UBYTE *)
        "MIDIDemo V2.0  ©'89/'90 N. Wirsing & I. Vassiliou für Markt & Technik",
        NULL,NULL,
        0,0,0,0,
        WBENCHSCREEN
};

             /***         Texte für den 'MyRequest()'-Aufruf            ***/
             /*** 'MyReqText' wird beim Aufruf der Funktion übergeben ! ***/

struct IntuiText MyReqText  = { FPEN,BPEN,JAM1,10,5,DFNT,NULL,0 };
struct IntuiText ReqGadText = { FPEN,BPEN,JAM1,5,3,DFNT,(UBYTE *)"Resume",0};

WORD MMoveFlag = MMF_NONE;

#define MAXVOL    0x3F        /*** Maximale Lautstärke für Amiga-Sounds ***/

#define ALL       0                         /*** Enable ALL MIDI CHANNELS ***/

WORD RcvChnl  = 1;                  /*** Enable ALL MIDI-RECEIVE-CHANNELS ***/
WORD SendChnl = 1;                  /***         MIDI-SEND-CHANNEL        ***/

struct SPNote SPNote;
UBYTE  SPVoices[4];
```

```
  /*** Amiga-Tasten zum Spielen von Instruments/Sounds (intern & MIDI !) ***/
static BYTE alphakeytable[128] = {
          0,    0, 0x31, 0x33,    0, 0x36, 0x38, 0x3A,
          0, 0x3D, 0x3F,    0, 0x42, 0x44,    0,    0,
       0x30, 0x32, 0x34, 0x35, 0x37, 0x39, 0x3B, 0x3C,
       0x3E, 0x40, 0x41, 0x43,    0,    0,    0,    0,
          0, 0x25, 0x27,    0, 0x2A, 0x2C, 0x2E,    0,
       0x31, 0x33,    0,    0,    0,    0,    0,    0,
          0, 0x24, 0x26, 0x28, 0x29, 0x2B, 0x2D, 0x2F,
       0x30, 0x32, 0x34,    0,    0,    0,    0,    0,
          0,    0,    0,    0,    0,    0,    0,    0,
          0,    0,    0,    0,    0,    0,    0,    0,
          0,    0,    0,    0,    0,    0,    0,    0,
          0,    0,    0,    0,    0,    0,    0,    0,
          0,    0,    0,    0,    0,    0,    0,    0,
          0,    0,    0,    0,    0,    0,    0,    0,
          0,    0,    0,    0,    0,    0,    0,    0,
          0,    0,    0,    0,    0,    0,    0,    0
};


USHORT PtrnData[] = {                /*** Daten des ZeichenMusters (Karo) ***/
        0x5555,0x5555,0x5555,0x5555,0x5555,0x5555,0x5555,0x5555,
        0x5555,0x5555,0x5555,0x5555,0x5555,0x5555,0x5555,0x5555,
        0x5555,0x5555,0x5555,0x5554,0xAAAA,0xAAAA,0xAAAA,0xAAAA,
        0xAAAA,0xAAAA,0xAAAA,0xAAAA,0xAAAA,0xAAAA,0xAAAA,0xAAAA,
        0xAAAA,0xAAAA,0xAAAA,0xAAAA,0xAAAA,0xAAAA,0xAAAA,0xAAA8
};

             /*** Das dazugehörige Image (2 Pixel hoch, 318 Pixel lang) ***/
struct Image PatternImage = { 0,0,318,2,2,&PtrnData[0],0x0001,0x0000,NULL };

                  /*** Dies sind die SpriteDaten des Window-Mauszeigers ***/
                                        /*** (37 Pixel hoch, 16 Pixel breit ***/
USHORT KeyboardPointer[]={
        0x0000, 0x0000,0xe000, 0xfe00,0x8000, 0xfc00,0x8000, 0x7800,0xf000, 0x0c00,
        0x8800, 0x0600,0x0400, 0x0300,0x0200, 0x0180,0x01c0, 0x0000,0x0000, 0x0000,
        0x3fff, 0x3fff,0x3fff, 0x3fff,0x3fff, 0x3f80,0x007f, 0x0000,0x3fff, 0x3f80,
        0x3fff, 0x3fff,0x3fff, 0x3f80,0x007f, 0x0000,0x3fff, 0x3f80,0x3fff, 0x3fff,
        0x3fff, 0x3fff,0x0000, 0x0000,0x3fff, 0x3fff,0x3fff, 0x3fff,0x3fff, 0x3f80,
        0x007f, 0x0000,0x3fff, 0x3f80,0x3fff, 0x3fff,0x3fff, 0x3f80,0x007f, 0x0000,
        0x3fff, 0x3f80,0x3fff, 0x3fff,0x3fff, 0x3f80,0x007f, 0x0000,0x3fff, 0x3f80,
        0x3fff, 0x3fff,0x3fff, 0x3fff,0x0000, 0x0000,0xffff, 0xffff,0x0000
};
```

Das Programm MIDIDemo.c

```
/*************************************************************************

        MIDIDemo.c (V1.0) © Nicolaus Wirsing 11.7.89 11:21
        Alle Rechte vorbehalten.
```

Taken from Amiga Manuals-Website

```
   MIDIDemo.c (V2.0)
   Modifiziert & verändert von Ilias Vassiliou 31.08.90
   © Ilias Vassiliou
       Alle Rechte vorbehalten.

   >>>>>>>>>>>>>>>>> For AZTEC-C V3.6 <<<<<<<<<<<<<<<<<<<<<<<
   >                                                       <
   >   cc +l -s +iMIDIDemo_Incl.pre MIDIDemo.c             <
   >   ln MIDIDemo.o Keyboard.o SMUSPlayer_Lib.o -lc32 +cd  <
   >                                                       <
   >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

   >   'MIDIDemo_Incl.pre' beinhaltet präcompilierte INCLUDE-Files
   >
   >   File : 'MIDIDemo_Incl.c'  :
   >
   >   #include <exec/types.h>
   >   #include <exec/memory.h>
   >   #include <devices/inputevent.h>
   >   #include <graphics/rastport.h>
   >   #include <graphics/gfxbase.h>
   >   #include <intuition/intuitionbase.h>
   >
   >   Compile : cc -s +l +hMIDIDemo_Incl.pre MIDIDemo_Incl.c

*************************************************************************/


/*** Falls nicht das präcompilierte INCLUDE-File benutzt wird, so müssen ***/
/*** an dieser Stelle die obigen '#include'-Angaben stehen !!!           ***/

#include "smusplayer_lib.h"
#include "midi.h"
#include "mididev.h"
#include "mididemo.h"

                          /*** Warten, bis Maustaste über einem Gadget  ***/
                          /*** wieder losgelassen wurde.                ***/
#define  WHILE_NOT_UP     while((class=Message(MIDIDemoWin))!=GADGETUP)

#define  ON    1
#define  OFF   0

#define  BOTH  4
#define  AMIG  5
#define  MIDI  6

struct IntuitionBase    *IntuitionBase  = NULL;
struct GfxBase          *GfxBase        = NULL;
struct SMUSPlayerBase   *SMUSPlayerBase = NULL;

struct IOMIDI           *OpenIOM        = NULL, *WriteIOM = NULL;
struct MsgPort          *RMPort         = NULL, *WMPort   = NULL;
```

Taken from Amiga-Manuals-Website

```
struct Window           *MIDIDemoWin     = NULL;
struct RastPort         *rp              = NULL;
struct TextFont         *MyFont          = NULL;
struct IntuiMessage     *msg             = NULL;

struct MIDIEvent        ReadME, WriteME;

WORD    kwd, oldnum, CloseFlag;

ULONG class,code,id;                  /*** IDCMP-Messages vom MIDIDemoWin ***/
BOOL  ReqAnswer=FALSE;                /***  Antwort von MyRequest-Aufruf  ***/
BYTE  As=OFF, Le=ON;            /*** Flags für ActSens & LocalEcho An/Aus ***/
BYTE  PlayFlag=BOTH;
UBYTE Program=0;
UBYTE *b;                           /*** Puffer-Byte für 'WToA()'-Aufruf ***/


      /**************************** FUNKTIONEN **************************/


VOID                      /*** AutoRequester mit übergebenem Text aufrufen ***/
MyRequest(MessageText)      /*** Erscheint auf dem gerade aktiven Fenster ***/
UBYTE *MessageText;
{
   MyReqText.IText = MessageText;          /*** übergebenen Text zuweisen ***/

   ReqAnswer = AutoRequest(IntuitionBase->ActiveWindow,&MyReqText,NULL,
          &ReqGadText, NULL,NULL,((LONG)strlen(MyReqText.IText)*8L+40L),50L);
}


VOID                /*** Libraries/Devices/IO/Windows schliessen, usw ... ***/
CloseAll(MessageText,ErrorNum)
UBYTE *MessageText;
WORD  ErrorNum;
{
   if(MessageText!=NULL) MyRequest(MessageText);

   if(OpenIOM!=NULL)
   {
          if(OpenIOM->IOM_IOStdR.io_Device)
                 CloseDevice(OpenIOM);
          DeleteExtIO(OpenIOM);
   }

   if(WriteIOM!=NULL)        DeleteExtIO(WriteIOM);
   if(RMPort!=NULL)          DeletePort (RMPort);
   if(WMPort!=NULL)          DeletePort (WMPort);
   if(MIDIDemoWin!=NULL)     CloseWindow(MIDIDemoWin);

   if(SMUSPlayerBase!=NULL) CloseLibrary(SMUSPlayerBase);
   if(GfxBase!=NULL)         CloseLibrary(GfxBase);
   if(IntuitionBase!=NULL)   CloseLibrary(IntuitionBase);
```

Taken from Amiga-Manuals-Website

```
   exit(ErrorNum);     /*** exit() muß hier stehen, sonst droht Absturz ! ***/
}


VOID                                /*** Libraries/Devices/IO ... öffnen ***/
OpenAll()
{
   if(!(IntuitionBase = (struct IntuitionBase *)
                            OpenLibrary("intuition.library",0L)))
   {
           puts("Konnte 'intuition.library' nicht öffnen !");
           CloseAll(NULL,20);
   }

   if(!(GfxBase = (struct GfxBase *)OpenLibrary("graphics.library",0L)))
        CloseAll("Konnte 'graphics.library' nicht öffnen !",20);

   if(!(SMUSPlayerBase = (struct SMUSPlayerBase *)
                             OpenLibrary("smusplayer.library",0L)))
        CloseAll("Konnte 'SMUSPlayer.library' nicht öffnen !",20);

                                       /*** zwei I/O-Requests vorbereiten ***/
   if(!(RMPort = CreatePort("MIDI-In",0L)))
        CloseAll("Konnte MIDI-In-MsgPort nicht erzeugen !!!",20);

   if(!(OpenIOM = (struct IOMIDI *)
             CreateExtIO(RMPort,sizeof(struct IOMIDI))))
        CloseAll("Konnte MIDI-IO-Block nicht erzeugen !!!",20);

   if(!(WMPort = CreatePort("MIDI-Out",0L)))
        CloseAll("Konnte MIDI-Out-MsgPort nicht erzeugen !!!",20);

   if(!(WriteIOM =(struct IOMIDI *)
             CreateExtIO(WMPort,sizeof(struct IOMIDI))))
        CloseAll("Konnte MIDI-IO-Block nicht erzeugen !!!",20);

                                                /*** MIDI.Device öffnen ***/
   OpenIOM->IOM_TypeFlags = METF_NOTEON|METF_NOTEOFF|METF_CCTRL|
                            METF_PITCHBEND|METF_ACTSENSON|METF_ACTSENSOFF;

   if(OpenDevice("midi.device",-1,OpenIOM,0L))
        CloseAll("Konnte 'midi.device' nicht öffnen !!!",20);

   WriteIOM->IOM_IOStdR.io_Device    =  OpenIOM->IOM_IOStdR.io_Device;
   WriteIOM->IOM_IOStdR.io_Unit      =  OpenIOM->IOM_IOStdR.io_Unit;
   WriteIOM->IOM_IOStdR.io_Command   =  MDCMD_WRITEMEVENT;
   WriteIOM->IOM_IOStdR.io_Data      =  &WriteME;
   WriteIOM->IOM_IOStdR.io_Length    =  sizeof(struct MIDIEvent);

   OpenIOM->IOM_IOStdR.io_Command    =  MDCMD_READMEVENT;
   OpenIOM->IOM_IOStdR.io_Data       =  &ReadME;
   OpenIOM->IOM_IOStdR.io_Length     =  sizeof(struct MIDIEvent);
   BeginIO(OpenIOM);
```

Taken from Amiga-Manuals-Website

```
        /*** Default-Instrument laden ( von Verzeichnis INSTRUMENTS: ) ***/
   if(LoadAux("INSTRUMENTS:default.instr") == -1)
      MyRequest("Konnte 'default.instr' nicht laden - ASSIGN INSTRUMENTS: !!!");
   SetDefault((BYTE *)0);        /*** Default Instrument = 0 (AUX) setzen ***/
}


VOID
WtoA(wert,string)        /***                    WORD to ASCII                ***/
WORD wert;               /*** Wandelt eine WORD-Zahl in das entsprechende ***/
BYTE *string;            /*** ASCII--quivalent um.                        ***/
{                        /*** Z.B.: Zahl : 23  wird zu String(!): "23"    ***/
   register WORD x,y,s;
   BYTE c;

   if((s=wert)<0)  wert=-wert;

   x=0;
   do
   {
      string[x++] = wert%10+'0';
   }while((wert/=10)>0);

   if(s<0) string[x++] = '-';
   string[x] = 0;
   for(x=0,y=(strlen(string)-1) ; x<y ; x++,y--)
   {
      c = string[x];
      string[x] = string[y];
      string[y] = c;
   }
}


ULONG                    /*** IDCMP-Message von beliebigen Window abrufen ***/
Message(win)
struct Window *win;
{
   struct IntuiMessage *m;
   ULONG  cl=0;                                    /*** Class-Variable ***/

   code=0;                  /*** 'code' ist hier eine GLOBALE VARIABLE, ***/
                            /*** der hier ein Wert zugewiesen wird !    ***/

   if(m=(struct IntuiMessage *)GetMsg(win->UserPort))
   {
      cl   = m->Class;      /*** 'cl' wird als Wert der Funtion 'Message' ***/
      code = m->Code;       /*** an den Aufrufer zurückgegeben !          ***/
      ReplyMsg(m);
   }
   return(cl);
}
```

Taken from Amiga-Manuals-Website

```
VOID                    /*** Text an beliebiger Position im Window ausgeben ***/
GEText(RP,text,color,mode,x,y)
struct RastPort *RP;                        /***   RastPort des Windows    ***/
STRPTR text;                                /***      Der Text selbst      ***/
UBYTE  color,mode;                          /***   Farbe und Zeichenmodus  ***/
WORD   x,y;                                 /*** Koordinaten der Eckpunkte ***/
{
   SetDrMd(RP,mode);                              /*** Zeichenmodus setzen ***/
   SetAPen(RP,color);                             /*** Zeichenfarbe setzen ***/
   Move(RP,x,y+(RP->TxBaseline));  /*** Zeichencursor an Position setzen ***/
   Text(RP,text,strlen(text));     /*** Jetzt Text in den RastPort malen ***/
}


VOID
TitleGad(win,gad,text)            /*** Ein beliebiges Gadget neu betiteln ***/
struct Window *win;                 /***        Adresse des Windows       ***/
struct Gadget *gad;                 /***        Adresse des Gadgets       ***/
struct IntuiText *text;             /*** Adresse einer IntuiText-Struktur ***/
{
   UWORD RealPosition;

   RealPosition = RemoveGList(win,gad,1);          /*** Gadget entfernen ***/
   gad->GadgetText = text;                         /***   Text zuweisen  ***/
                                                   /*** Gadget einfügen  ***/
   RealPosition = AddGList(win,gad,RealPosition,1,NULL);
   RefreshGList(gad,win,NULL,1);                   /*** Gadget zeichnen  ***/
}


VOID                              /***   Ton spielen = Taste markieren   ***/
PlayKey(num,vol)                  /*** über SMUSPlayer und MIDI ausgeben ***/
UWORD num, vol;
{
        register WORD n;

   if( (PlayFlag==BOTH) || (PlayFlag==AMIG) )            /*** Flag testen ***/
   {
        SPNote.iName    =  SP_DEFAULTINAME;
        SPNote.SNote    =  num<<8;
        SPNote.vol      =  vol<<2;
        SPNote.voice    =  SP_DYNAMICVOICE;
        SPNote.pri      =  SP_PLAYERPRI+1;

        n = PlayTone(&SPNote);
        if(n != SP_ERROR) SPVoices[n-1]  =  num;
   }

   if( (PlayFlag==BOTH) || (PlayFlag==MIDI) )            /*** Flag testen ***/
   {
        WriteIOM->IOM_IOStdR.io_Command  =  MDCMD_WRITEMEVENT;
        WriteME.me_Status                =  MS_NOTEON;
```

Taken from Amiga-Manuals-Website

```
        WriteME.me_Channel              =  SendChnl-1;
        WriteME.me_Data1                =  num;
        WriteME.me_Data2                =  127;
        DoIO(WriteIOM);
    }

    PaintKey(num,SLCTKEYCOLOR1);       /*** Keyboardtaste im Window bemalen ***/
}


VOID
ReleaseKey(num)                        /*** Ton (intern/MIDI) deaktivieren ***/
UWORD num;
{
    register WORD i;

    if( (PlayFlag==BOTH) || (PlayFlag==AMIG) )          /*** Flag testen ***/
    {
        for(i=0; i<4; i++)
        {
                if(SPVoices[i] == num)
                {
                        SPVoices[i] = 0;
                        ReleaseTone(i+1);
                        break;
                }
        }
    }

    if( (PlayFlag==BOTH) || (PlayFlag==MIDI) )          /*** Flag testen ***/
    {
        WriteIOM->IOM_IOStdR.io_Command  =  MDCMD_WRITEMEVENT;
        WriteME.me_Status               =  MS_NOTEON;
        WriteME.me_Channel              =  SendChnl-1;
        WriteME.me_Data1                =  num;
        WriteME.me_Data2                =  0;
        DoIO(WriteIOM);
    }

    PaintKey(num, NORMKEYCOLOR);       /*** Keyboardaste normal anzeigen ***/
}


VOID                            /*** Hier werden die Maus-Clicks behandelt ***/
ProcessMouse(x,y,co)
WORD x,y,co;
{
    register WORD num;

    if(num=KeyNum(x,y))                /*** Keyboard-TastenNummer ermitteln ***/
    {
        if(co == SELECTDOWN)           /***    Linke MausTaste gedrückt?    ***/
        {
```

Taken from Amiga-Manuals-Website

```
         MMoveFlag = MMF_KEYBOARD;
         oldnum    = num;
         PlayKey(num, MAXVOL);          /***         Ton spielen          ***/
      }
      else if(co == SELECTUP)           /*** Linke MausTaste losgelassen? ***/
      {
         MMoveFlag = MMF_NONE;
         oldnum    = 0;
         ReleaseKey(num);               /***     Tonausgabe abbrechen     ***/
      }
   }
   else if( (oldnum != 0) && (co == SELECTUP) )
   {
      MMoveFlag = MMF_NONE;     /*** Falls neue Taste ermittelt wurde,     ***/
      ReleaseKey(oldnum);       /*** ist die Tonausgabe für die vorherige ***/
      oldnum = 0;               /***           Taste abzubrechen          ***/
   }
}


VOID                                    /*** Mausbewegungen bearbeiten ***/
ProcessMMove(x,y)
WORD x,y;
{
   register UWORD num,v;

   switch(MMoveFlag){
        case MMF_NONE     : break;
        case MMF_KEYBOARD :{
                        /*** Taste des Keyboards bestimmen & Ton spielen ***/
                        if((num=KeyNum(x,y)) && (num!=oldnum) )
                        {
                           ReleaseKey(oldnum);  /*** alten Ton loslassen ***/
                           PlayKey(num, MAXVOL); /***    neuen spielen   ***/
                           oldnum = num;       /** neuer Ton = alter Ton ***/
                        }
                        break;
                     }
        case MMF_PITCHWHEEL:{
                        WriteIOM->IOM_IOStdR.io_Command  =  MDCMD_WRITEMEVENT;
                        WriteME.me_Status                =  MS_PITCHBEND;
                        WriteME.me_Channel               =  SendChnl-1;

                        v = (MAXPOT-(((struct PropInfo *)
                        (G(PITWID)->SpecialInfo))->VertPot))>>2;

                        WriteME.me_MsbLsb = v;
                        DoIO(WriteIOM);
                        GetMsg(WMPort);
                        break;
                     }
```

Taken from Amiga-Manuals-Website

```
                case MMF_MODWHEEL:{
                        WriteIOM->IOM_IOStdR.io_Command  = MDCMD_WRITEMEVENT;
                        WriteME.me_Status                = MS_CTRL;
                        WriteME.me_Channel               = SendChnl-1;
                        WriteME.me_Data1                 = MC_MODWHEEL;

                        WriteME.me_MsbLsb  = (MAXPOT-(((struct PropInfo *)
                              (G(MODWID)->SpecialInfo))->VertPot))>>2;

                         DoIO(WriteIOM);
                         GetMsg(WMPort);
                         break;
                       }
   }
}


VOID                      /*** ProgramChange-Meldung an MIDI-Gerät senden ***/
DoProgChange(prog)
UBYTE prog;
{
   WriteIOM->IOM_IOStdR.io_Command  =  MDCMD_WRITEMEVENT;
   WriteME.me_Status                =  MS_PROG;
   WriteME.me_Channel               =  SendChnl-1;
   WriteME.me_Data1                 =  prog-1;
   WriteME.me_Data2                 =  0;
   WriteME.me_MsbLsb                =  -1;
   DoIO(WriteIOM);
   GetMsg(WMPort);
}


VOID                          /*** Text in Integer Gadgets verändern ***/
SetIGText(GId,StrInfoNum,k)
UWORD GId,StrInfoNum,k;
{
   if(k != ALL)
   {
      WtoA(k,b);                          /*** Zahl in String umwandeln ***/
      strcpy(ChString[StrInfoNum].Buffer,b); /*** In Gad.Puffer kopieren ***/
   }
   else
   {
      ChString[StrInfoNum].Buffer[0]   = 'A';
      ChString[StrInfoNum].Buffer[1]   = 'L';
   }

   RefreshGList(G(GId),MIDIDemoWin,0,1);
}
```

Taken from Amiga-Manuals-Website

```
VOID                                                    /*** Gadgets bearbeiten ***/
ProcessGadgets(co)
WORD co;
{
   switch(co)
   {
         case RCHUID:{              /***     RcvChnl-Up-PfeilGadget     ***/
                  WHILE_NOT_UP  /*** Solange nicht GADGETUP empfangen ***/
                  {
                     if(RcvChnl<16)                /***    Hochzählen    ***/
                     {
                        RcvChnl++;                 /*** Wert darstellen ***/
                        SetIGText(RCHSID,0,RcvChnl);
                     }
                     Delay(5);    /*** damit es nicht zu schnell geht ***/
                  }
                  break;
               }
         case RCHDID:{                     /*** RcvChnl-Down-PfeilGadget ***/
                  WHILE_NOT_UP
                  {
                     if(RcvChnl>ALL)               /***  Herunterzählen ***/
                     {
                        RcvChnl--;                 /*** Wert darstellen ***/
                        SetIGText(RCHSID,0,RcvChnl);
                     }
                     Delay(5);                     /***    wie oben     ***/
                  }
                  break;
               }
         case SCHUID:{                     /*** SendChnl-Up-PfeilGadget ***/
                  WHILE_NOT_UP
                  {
                     if(SendChnl<16)
                     {
                        SendChnl++;
                        SetIGText(SCHSID,1,SendChnl);
                     }
                     Delay(5);
                  }
                        break;
               }
         case SCHDID:{                     /*** SendChnl-Down-PfeilGadget ***/
                  WHILE_NOT_UP
                  {
                     if(SendChnl>1)
                     {
                        SendChnl--;
                        SetIGText(SCHSID,1,SendChnl);
                     }
                     Delay(5);
                  }
```

Taken from Amiga-Manuals-Website

```
                    break;
                }
            case PGCHDWN:{                  /*** PrgChange-Down-PfeilGadget ***/
                    WHILE_NOT_UP
                    {
                        if(Program>1)
                        {
                            Program--;
                            SetIGText(PGCHNGE,2,Program);
                        }
                        Delay(5);
                    }
                    DoProgChange(Program);
                    break;
                }
            case PGCHUP:{                     /*** PrgChange-Up-PfeilGadget ***/
                    WHILE_NOT_UP
                    {
                        if(Program<128)
                        {
                            Program++;
                            SetIGText(PGCHNGE,2,Program);
                        }
                        Delay(5);
                    }
                    DoProgChange(Program);
                    break;
                }
            case PITWID:{                           /*** PitchProp-Gadget ***/
                    MMoveFlag = MMF_PITCHWHEEL;
                    break;
                }
            case MODWID:{                           /***  ModProp-Gadget  ***/
                    MMoveFlag = MMF_MODWHEEL;
                    break;
                }
        }
}


VOID            /*** ActSens o. LocalEcho ON/OFF über midi.device senden ***/
SendOnOff(gad,flag,on_cmd,off_cmd)
struct Gadget *gad;             /***  Gadget, dessen Text verändert wird  ***/
BYTE *flag;                     /***                ON/OFF - Flag         ***/
UWORD on_cmd,off_cmd;           /*** ON/OFF Commandos f.r das midi.device ***/
{
    if(*flag==OFF)
    {
        *flag=ON;                                   /***    Flag kippen   ***/
        WriteIOM->IOM_IOStdR.io_Command = on_cmd;   /*** Commando setzen  ***/
        goto DOIT;
    }
```

Taken from Amiga-Manuals-Website

```
   if(*flag==ON)
   {
      *flag=OFF;                                    /***    wie oben     ***/
      WriteIOM->IOM_IOStdR.io_Command = off_cmd;
   }

DOIT:
   TitleGad(MIDIDemoWin,gad,&MIDIWinText[*flag]);   /*** gad-Text ändern ***/
   DoIO(WriteIOM);                                 /***  Commando abschicken  ***/
   GetMsg(WMPort);                                 /*** auf Rückmeldung warten ***/
}


VOID                                        /*** GadgetUP-Messages bearbeiten ***/
ProcessGadgetUP(co)
WORD co;
{
   switch(co)
   {
      case ASONID:{                                 /***  ActSens-Gadget ***/
               SendOnOff(G(0),&As,MDCMD_ACTSENSON,MDCMD_ACTSENSOFF);
               break;
            }
      case LEONID:{                                 /*** LocalEcho-Gadget ***/
               SendOnOff(G(1),&Le,MDCMD_LOCALECHOON,MDCMD_LOCALECHOOFF);
               break;
            }
      case RCHSID:{                             /*** RcvChnl-StringGadget ***/
               if( (RCHS->LongInt > -1) && (RCHS->LongInt < 17) )
                     RcvChnl = RCHS->LongInt;      /***  Wert übernehmen  ***/
               else
                     SetIGText(RCHSID,0,RcvChnl);
               if(RcvChnl == 0)                    /***       0 = ALL     ***/
                     SetIGText(RCHSID,0,RcvChnl); /***  'AL' darstellen  ***/
               break;
            }
      case SCHSID:{                            /*** SendChnl-StringGadget ***/
               if( (SCHS->LongInt>0) && (SCHS->LongInt<17) )
                     SendChnl = SCHS->LongInt;             /*** wie oben ***/
               else
                     SetIGText(SCHSID,1,SendChnl);
               break;
            }
      case PGCHNGE:{                           /*** PGChange-StringGadget ***/
            if( (PCHS->LongInt>1) && (PCHS->LongInt<129) )
                     Program = PCHS->LongInt;              /*** wie oben ***/
            else
                     SetIGText(PGCHNGE,2,Program);
            DoProgChange(Program);
            break;
            }
```

```
      case ENPLID:{                                  /*** EnablePlay-Gadget ***/
             if(PlayFlag<MIDI) PlayFlag++;                /*** Flag testen ***/
                    else PlayFlag=BOTH;                   /*** und setzen  ***/
                                  /*** Gadget je nach Flag-Status betiteln ***/
              TitleGad(MIDIDemoWin,G(10),&MIDIWinText[PlayFlag]);
              break;
            }
      case PITWID:                            /*** Pitch- & Mod-Prop-Gadgets ***/
      case MODWID:{
              MMoveFlag = MMF_NONE;
              break;
            }
   }
}


                          /*** RawKey-Amiga-Tastatur-Messages bearbeiten ***/
ProcessKey(co)
UWORD co;
{
   BYTE num;

   if((num = alphakeytable[co & 0x7f]) > 0)
   {                                                 /***   Taste gedrückt  ***/
      if(!(co & IECODE_UP_PREFIX)) PlayKey(num,MAXVOL);
      else ReleaseKey(num);                          /*** Taste losgelassen ***/
   }
}



      /************************** HAUPTPROGRAMM **********************/


VOID
main()
{
   register ULONG WaitFlags;
   register SHORT xx,yy;

   OpenAll();                                            /*** Alles öffnen ***/

                                  /*** Window öffnen und initialisieren ***/
   if(!(MIDIDemoWin = (struct Window *)OpenWindow(&MIDIDemoWindow)))
        CloseAll("Konnte MIDIDemoWindow nicht öffnen !!!",20);

   /*** Ein bißchen 'Malen nach Zahlen', weil's professioneller aussieht ***/

   rp = MIDIDemoWin->RPort;          /*** RastPort des Fensters bestimmen ***/

   for(yy=10;yy<84;yy+=2)               /*** Hintergrund des Fensters mit ***/
   {                                    /*** 'kleinkariertem' Pattern     ***/
```

Taken from Amiga-Manuals-Website

```
   for(xx=2;xx<630;xx+=318)          /*** (Zeichenmuster) bemalen       ***/
       DrawImage(rp,&PatternImage,xx,yy);
}

SetDrMd(rp,JAM2);              /***          Zeichenmodus setzen          ***/
SetAPen(rp,2);                 /***   Zeichenfarbe (schwarz) setzen       ***/
RectFill(rp,11,12,629,30);     /*** Oberes schwarzes Rechteck  zeichnen ***/
RectFill(rp,154,32,482,46);    /***   EnablePlay Hintergrund-Rechteck    ***/

AddGList(MIDIDemoWin,G(0),0,-1,NULL);   /***     Gadgets anhängen         ***/
RefreshGadgets(G(0),MIDIDemoWin,NULL); /*** Alle Gadgets neu zeichnen ***/

       /*** Man sollte sich nie darauf verlassen, daß der System-Font ***/
       /*** immer TOPAZ_80 ist - deshalb explizite Anforderung !!!    ***/

if( (MyFont = OpenFont(DFNT)) == NULL)
    MyRequest("Konnte DefaultFont nicht öffnen !");
else
    SetFont(rp,MyFont);            /*** Font f. d. RastPort setzen !!! ***/

GEText(rp,"Active Sensing",FPEN,JAM1,13,17);  /*** Window-Texte malen ***/
GEText(rp,"Local Echo"    ,FPEN,JAM1,175,17);
GEText(rp,"AS off"        ,BPEN,JAM2,ASTEXTL,ASTEXTT);
GEText(rp,"Pitch"         ,BPEN,JAM2,13,39);
GEText(rp,"Mod"           ,BPEN,JAM2,60,39);
GEText(rp,"ProgramChange" ,FPEN,JAM1,309,36);

                                          /*** Window-Mauszeiger setzen ***/
SetPointer(MIDIDemoWin,&KeyboardPointer[0],37,16,0,0);

kwd = InitKeyBoard(rp,KEYBOARDL,KEYBOARDT);    /*** Keyboard zeichnen ***/


                                 /*** Bis Click auf Close-Gadget warten ***/
while(!CloseFlag)
{
                  /*** Auf Eingabe vom Window oder MIDI-Meldung warten ***/
   WaitFlags = (Wait(1<<MIDIDemoWin->UserPort->mp_SigBit |
           1<<RMPort->mp_SigBit) );

                                  /*** Eingabe vom Window auswerten ***/
   if(WaitFlags & 1<<MIDIDemoWin->UserPort->mp_SigBit)
   {
        while(msg=(struct IntuiMessage *)GetMsg(MIDIDemoWin->UserPort))
        {
             switch(msg->Class)
             {
                  case MOUSEBUTTONS:{   /***  MausTasten gedrückt    ***/
                       ProcessMouse(msg->MouseX,msg->MouseY,
                            msg->Code);
                       break;
                  }
```

Taken from Amiga-Manuals-Website

```
                    case MOUSEMOVE:{     /*** MausBewegung registriert ***/
                         ProcessMMove(msg->MouseX,msg->MouseY);
                         break;
                       }
                    case GADGETDOWN:{  /*** GadgetDown-Betätigung registriert ***/
                         ProcessGadgets(((struct Gadget *)
                               msg->IAddress)->GadgetID);
                         break;
                       }
                    case GADGETUP:{      /*** GadgetUp-Betätigung registriert ***/
                         ProcessGadgetUP(((struct Gadget *)
                               msg->IAddress)->GadgetID);
                         break;
                       }
                    case RAWKEY:{     /*** Amiga-Tasten-Betätigung registriert ***/
                         if(!(msg->Qualifier & IEQUALIFIER_REPEAT))
                               ProcessKey(msg->Code);
                         break;
                       }
                    case CLOSEWINDOW:{    /*** CloseGadget registriert ***/
                         CloseFlag = 1;
                         break;
                       }
               }
               ReplyMsg(msg);  /*** Empfangene Message beantworten - wichtig ! ***/
          }
     }

     if(WaitFlags & 1<<RMPort->mp_SigBit)      /*** MIDIEvents auswerten ***/
     {
        do
        {
           GetMsg(RMPort);                     /*** MIDI-Message abholen ***/
           if( (RcvChnl==ALL) || (RcvChnl==(ReadME.me_Channel+1)) )
           {

          /***  Im Struktur-Eintrag 'ReadME.me_Status' steht die Art der ***/
          /***  empfangenen MIDI-Nachricht - definiert im Include-File   ***/
          /***  'midi.h' unter 'Channel Voice Messages'.                 ***/
          /***  Diese Nachrichten haben das Format: 1sssnnnn.            ***/
          /***  In den vier nnnn-Bits steht folglich die MIDI-Kanalnum-  ***/
          /***  mer; aus diesem Grunde wird unter switch(...) der Wert   ***/
          /***  von 'ReadME.me_Channel' subtrahiert, um die reinen Stati ***/
          /***  zu erhalten, nach denen in den jeweiligen 'case' nun     ***/
          /***  verzweigt werden kann.                                   ***/

             switch(ReadME.me_Status-ReadME.me_Channel)

               case MS_NOTEOFF:{  /*** MIDIKeyboard-Taste losgelassen ***/
                       ReleaseKey(ReadME.me_Data1);
                       break;
                     }
```

Taken from Amiga-Manuals-Website

```
                case MS_NOTEON:{  /*** MIDIKeyboard-Taste angeschlagen ***/
                        if(ReadME.me_Data2)
                             PlayKey(ReadME.me_Data1,ReadME.me_Data2);
                             /*** Falls Dynamik==0, Taste losgelassen ***/
                        else ReleaseKey(ReadME.me_Data1);
                        break;
                      }
                case MS_CTRL:{        /*** MIDI-ControlChange empfangen ***/
                        if(ReadME.me_Data1 == MC_MODWHEEL)
                             NewModifyProp(G(MODWID),MIDIDemoWin,0,
                                        AUTOKNOB|FREEVERT,PPHPOT,
                                        MAXPOT-(ReadME.me_MsbLsb<<2),
                                        PPHBODY, PPVBODY,1);
                        break;
                      }
                case MS_PITCHBEND:{/*** MIDI-PitchBendChange empfangen ***/
                        NewModifyProp(G(PITWID), MIDIDemoWin,
                                    0,AUTOKNOB|FREEVERT,PPHPOT,
                                    MAXPOT-(ReadME.me_MsbLsb<<2),
                                    PPHBODY, PPVBODY,1);
                        break;
                      }
            }
        }

        switch(ReadME.me_Status)
        {
            case MPS_ACTSENSON:{     /*** MIDI-ACTIVESENS_ON empfangen ***/
                    GEText(rp,"AS on ",3,JAM2,ASTEXTL,ASTEXTT);
                    break;
                  }
            case MPS_ACTSENSOFF:{    /*** MIDI-ACTIVESENS_OFF empfangen ***/
                    GEText(rp,"AS off",1,JAM2,ASTEXTL,ASTEXTT);
                    break;
                  }
        }
        SetSignal(0,1<<RMPort->mp_SigBit);  /*** Signal-Maske setzen ***/
        OpenIOM->IOM_IOStdR.io_Flags |= IOF_QUICK;
        BeginIO(OpenIOM);
      } while(OpenIOM->IOM_IOStdR.io_Flags & IOF_QUICK);

    }                                /***       if(WaitFlags & ...)    ***/
  }                                  /***         While (!Close)       ***/
  FreeKeyBoard();                    /*** Keyboard-Speicher freigeben ***/
  CloseAll(NULL,0);                  /***           aufräumen          ***/
}
```

Taken from Amiga-Manuals-Website

Das Programm keyboard.c

```
/*****************************************************************************

        keyboard.c (c) Nicolaus Wirsing 11.7.89

        All Rights reserved.

   Modified by Ilias Vassiliou 31.08.90

   ***************** For AZTEC-C V3.6 *******************

   cc -e2048 +l -s +iMIDIDemo_Incl.pre Keyboard.c
   ln MIDIDemo.o Keyboard.o SMUSPlayer_Lib.o -lc32

   ****************************************************


*****************************************************************************/

                    /*** Keyboard muß auf Ganztönen beginnen und enden! ***/
#define LOWTONE             36
#define HITONE              108

#define KEYSPEROCT          12
#define WKEYSPEROCT         7

struct SKey {
        UBYTE pos;
        UBYTE type;
};

                   /*** 5 Tastenumrisse: weiß, rechts schwarz, links schwarz,
                                   rechts und links schwarz, schwarz  ***/
#define WHITEKEY            0
#define RBLACKKEY           1
#define LBLACKKEY           2
#define RLBLACKKEY          3
#define BLACKKEY            4
#define KEYTYPES            5

STATIC struct SKey keytable[12] = {
        0,RBLACKKEY,                                        /*** C  ***/
        0,BLACKKEY,                                         /*** C# ***/
        1,RLBLACKKEY,                                       /*** D  ***/
        1,BLACKKEY,                                         /*** D# ***/
        2,LBLACKKEY,                                        /*** E  ***/
        3,RBLACKKEY,                                        /*** F  ***/
        3,BLACKKEY,                                         /*** F# ***/
        4,RLBLACKKEY,                                       /*** G  ***/
        4,BLACKKEY,                                         /*** G# ***/
```

Taken from Amiga-Manuals-Website

```
        5,RLBLACKKEY,                                           /*** A  ***/
        5,BLACKKEY,                                             /*** Bb ***/
        6,LBLACKKEY                                             /*** H  ***/
};

#define KLCP(p)             ((p)%WKEYSPEROCT)
#define KEYNUM(pos)         (((WORD)((pos)/WKEYSPEROCT)*KEYSPEROCT)+\
                            (KLCP(pos)*2)-(KLCP(pos)<3?0:1))

#define WKEYPOS(n)          (((n)/2)+((n)<5?0:1))


                    /*** Höhe und Breite der schwarzen und weißen Tasten ***/
#define WHITEHT             30
#define WHITEWD             12
#define BLACKHT             15
#define BLACKWD             7
#define LOWC(n)             ((WORD)((n)/KEYSPEROCT))
#define KEYBOARDX(k)        (WHITEWD*((LOWC(k)-LOWC(LOWTONE))*WKEYSPEROCT+\
                            keytable[(k)%KEYSPEROCT].pos-keytable[LOWTONE%\
                            KEYSPEROCT].pos))

#define SHAPEWD             (WHITEWD+(WORD)((BLACKWD+1)/2))
#define BLACKEDWHITEWD      (WHITEWD-(WORD)(BLACKWD/2))
#define BYTESIZE(x)         (((WORD)((x)/16)+1)*2)
#define BYTESPERLINE        BYTESIZE(SHAPEWD)
#define KEYSHAPESIZE        (BYTESIZE(SHAPEWD)*WHITEHT)

#define POS(x)              ((x)<0?0:(x))
#define LSETMASK(n)         (~0<<POS(n))
#define RSETMASK(n)         (~0^(~0<<POS(n)))
#define DRAWMASK(b,lx,rx)   ((unsigned char)(RSETMASK((b+1)*8-(lx))&\
                            (LSETMASK((b+1)*8-(rx))) ))

#define KEYARRAYSIZE        (KEYSHAPESIZE*KEYTYPES)

#define NORMKEYCOLOR        0               /*** Farbnummern für PaintKey ***/
#define SLCTKEYCOLOR1       1
#define SLCTKEYCOLOR2       2

struct Shape {
        UBYTE data[WHITEHT][BYTESPERLINE];
};


STATIC struct  GfxBase      *GfxBase;
STATIC struct  RastPort     *DrawRP;
STATIC struct  Shape        *keyshapes = NULL;

STATIC WORD    keybX, keybY, keybW;
```

Taken from Amiga-Manuals-Website

```
     /*** Intialisiert und zeichnet Keyboard an der Position posx, posy ***/
WORD
InitKeyBoard(rp,posx,posy)
struct RastPort *rp;
WORD posx, posy;
{
   WORD shi, li, bi;
   struct Shape *ks;

   DrawRP = rp;
   keybX = posx;
   keybY = posy;
   keybW = KEYBOARDX(HITONE+1);
                                        /*** Graphics.library öffnen ***/
   if(!(GfxBase=(struct GfxBase *)OpenLibrary("graphics.library",0)))
           return(0);

                            /*** Speicher für Tastenumrisse reservieren ***/
   if(!(keyshapes = (struct Shape *)AllocMem(KEYARRAYSIZE,
        MEMF_PUBLIC|MEMF_CHIP|MEMF_CLEAR)) )
   {
           CloseLibrary(GfxBase);
           return(0);
   }

   ks = keyshapes;

                        /*** Tastenumrisse aus Höhe und Breite berechnen ***/
   for(shi=0;shi<KEYTYPES;shi++,ks++)
   {
       for(li=0;li<WHITEHT;li++)
       {
           for(bi=0;bi<BYTESPERLINE;bi++)
           {
               switch(shi){
                   case WHITEKEY:{
                        ks->data[li][bi] = DRAWMASK(bi,0,WHITEWD-1);
                        break;
                      }
                   case RBLACKKEY:{
                        ks->data[li][bi] = (li<BLACKHT) ?
                           DRAWMASK(bi,0,BLACKEDWHITEWD):
                              DRAWMASK(bi,0,WHITEWD-1);
                        break;
                      }
                   case LBLACKKEY:{
                        ks->data[li][bi] = (li<BLACKHT) ?
                           DRAWMASK(bi,(WORD)(BLACKWD/2),WHITEWD-1):
                              DRAWMASK(bi,0,WHITEWD-1);
                        break;
                      }
```

Taken from Amiga-Manuals-Website

```
                    case RLBLACKKEY:{
                         ks->data[li][bi] = (li<BLACKHT) ?
                            DRAWMASK(bi,(WORD)(BLACKWD/2),BLACKEDWHITEWD):
                               DRAWMASK(bi,0,WHITEWD-1);
                         break;
                       }
                    case BLACKKEY:{
                         ks->data[li][bi] = (li<BLACKHT) ?
                            DRAWMASK(bi,BLACKEDWHITEWD,SHAPEWD) : 0;
                         break;
                       }
                 }
            }
        }
    }

    SetAPen(DrawRP, 1);                          /*** Keyboard zeichnen ***/
    SetDrMd(DrawRP, JAM1);
    RectFill(DrawRP, keybX, keybY, keybX+keybW-1,
    keybY+WHITEHT-1);

    SetAPen(DrawRP,2);
    for(li=keybX-1; li<keybX+keybW; li+=WHITEWD)
    {
            Move(DrawRP, li, keybY);
            Draw(DrawRP, li, keybY+WHITEHT);
    }

    for(li=LOWTONE;li<HITONE;li++)
    {
        if(keytable[li%KEYSPEROCT+1].type == BLACKKEY)
        {
              BltTemplate(&(keyshapes[BLACKKEY]), 0, BYTESPERLINE,
              DrawRP, keybX+KEYBOARDX(li), keybY, SHAPEWD, WHITEHT);
        }
    }
    return(keybW);
}


VOID                                     /*** Speicher zurückgeben etc. ***/
FreeKeyBoard()
{
    FreeMem(keyshapes,KEYARRAYSIZE);
    CloseLibrary(GfxBase);
}


VOID                    /*** Taste normal oder highlighted nachzeichnen ***/
PaintKey(n, color)
WORD n, color;
{
    REGISTER WORD c;
```

Taken from Amiga-Manuals-Website

```
    if(n>=LOWTONE &&n<=HITONE)
    {
        c = color ? (color == SLCTKEYCOLOR1 ? 3 : 0) :
           (keytable[n%KEYSPEROCT].type == BLACKKEY ? 2 : 1);
        SetDrMd(DrawRP, JAM1);
        SetAPen(DrawRP, c);
        BltTemplate(&(keyshapes[keytable[n%KEYSPEROCT].type]),0, BYTESPERLINE,
        DrawRP, keybX+KEYBOARDX(n), keybY, SHAPEWD, WHITEHT);
    }
}


UWORD      /*** Tonnummer zu einer Position im RastPort (0 = ausserhalb) ***/
KeyNum(x,y)
UWORD x, y;
{
    REGISTER UWORD xpos, ypos, p, t;

    xpos = x-keybX;
    ypos = y-keybY;

    if(xpos<keybW && ypos<WHITEHT)
    {
        p = LOWTONE+KEYNUM((WORD)(xpos/WHITEWD));
        t = keytable[p%KEYSPEROCT].type;
        return( p+ ( ypos>BLACKHT ? 0 :
          ((t & LBLACKKEY) != 0 && xpos%WHITEWD <= ((WORD)(BLACKWD/2)+1) &&
               xpos >= WHITEWD*1 ? (-1) :
          ((t & RBLACKKEY) != 0 && xpos%WHITEWD > (WHITEWD-(WORD)(BLACKWD/2)) &&
               xpos < keybW-WHITEWD*1 ? 1: 0))) );
    }
    else return(0);

}
```

5.3 MIDI-kompatible Musikinstrumente

Wie schon am Beginn dieses Kapitels erwähnt, hat die MIDI-Schnittstelle seit ihrer Einführung die Tonstudios der Welt im Sturm erobert. Obwohl sie ursprünglich nur für Keyboarder zur Verbindung von Synthesizern konzipiert wurde, sind inzwischen für fast alle Instrumente entsprechende Controller vorhanden, mit denen der Musiker »Anschluß« an die faszinierende Welt der MIDI-kompatiblen Geräte findet. Bei der Wahl des Instruments, welches man über seinen Controller spielen möchte, ist man jedoch im allgemeinen auf elektronische Wiedergabemedien angewiesen: Da eine Schnittstelle zur Datenbehandlung einen Computer benötigt, der in diesem Fall ja auch den Klanggenerator zu steuern hat, werden heutzutage so gut wie alle Synthesizer digital gesteuert. Theoretisch könnte man zwar auch mechanische Instrumente für die Wiedergabe nachrüsten, mit vertretbarem Aufwand jedoch lassen sich Instrumente mit akzeptabler Flexibilität nur rein elektronisch realisieren.

Tandy Amiga Manuals-Website

In diesem Abschnitt soll zunächst eine Übersicht über die gängigen Syntheseverfahren und ihre Grundprinzipien gegeben werden. Anschließend werden einige der beliebtesten MIDI-Geräte kurz vorgestellt. Diese werden aufgegliedert in Keyboard-Synthis, Expander und Drum-Sequencer.

5.3.1 Syntheseverfahren

Seit dem Erscheinen des Urvaters aller Synthesizer, des MOOG, wurde eine ganze Reihe verschiedener Varianten der elektronischen Klangsynthese entwickelt, deren Hauptintention es stets ist, eine Vielzahl von Parametern und damit einen möglichst großen Klangreichtum zur Verfügung zu stellen. Gleichzeitig sollte in einem Verfahren aber auch ein einfacher Zusamenhang zwischen allen vorhandenen Parametern und dem Ergebnis, also dem Klang, bestehen. Dieser Zusammenhang sollte genug Übersicht über die Effekte der einzelnen Änderungen bieten, damit ein gewünschter Sound auf einfache Weise konstruiert werden kann. Und als drittes Kriterium ist wohl auch der Hardwareaufwand zu nennen, der für die Realisierung eines bestimmten Syntheseverfahrens nötig ist. Wie viele Tongeneratoren in einem Synthesizer untergebracht werden können, wird nicht zuletzt vom Preis eines Generators bestimmt. Die Anzahl der Tongeneratoren ist jedoch vielleicht die wichtigste Eigenschaft eines Gerätes, da die Vielfalt der möglichen Sounds und Effekte direkt von ihr abhängig ist.

Analoge Syntheseverfahren

Vor allem die älteren Geräte, zu deren Zeit die Bauteilintegration noch nicht so weit fortgeschritten war, bedienen sich dieser Technik. Bei diesem Verfahren werden die Basiswellenformen analog erzeugt, d.h. durch einen einfachen Oszillator. Solche analogen Oszillatoren sind aber nur zur Erzeugung von sehr wenigen Wellenformen imstande, also meist nur Sinus, Rechteck, Dreieck und Sägezahn. Ihre oft doch beeindruckende Klangfülle erreichen diese Synthesizer durch den gleichzeitigen Einsatz mehrerer Tongeneratoren sowie mit Hilfe einer komplexen Amplituden- und Filtermodulation.

Auch heute werden teilweise noch Analogsynthesizer gebaut, zum einen weil mit diesem Prinzip sehr volle, warme Klangbilder erzeugt werden können, zum anderen weil auch hier der Fortschritt nicht haltgemacht hat und analoge Tongeneratoren mittlerweile preisgünstig realisierbar sind.

PCM-Synthese

Die einfachste Form der digitalen Tonerzeugung bietet die sog. PCM-(Pulse-Code-Modulation-) Synthese. Dies entspricht der einfachen Ausgabe von Sample-Daten an einen D/A-Wandler, wobei die Oberwellenverteilung allein von den ausgegebenen Daten abhängig ist. Hierbei sind grundsätzlich zwei Typen zu unterscheiden: zum einen die Ausgabe der Daten eines kompletten akustischen Ereignisses, wobei dieses nur bezüglich seiner Lautstärke und Tonhöhe verändert werden kann. Auf diese Weise arbeiten die sog. Sampler, die lediglich lange Samplefolgen auf Befehl zum D/A-Wandler ausgeben, und in manchen Fällen auch digital mischen können.

Die zweite Anwendung der PCM-Synthese ist die Erzeugung von beliebig komplexen Wellenformen, die durch ständige Wiederholung eines bestimmten Sample-Datenfeldes erzeugt werden. Hier kann durch die Kombination vieler verschiedener Wellenformen und individuelle Hüllkurven- und Filtermodulation eine ansonsten unerreichte Vielfalt von verschiedenen Klangcharakteristiken erzielt werden.

Additive Synthese

Diese Syntheseform beruht auf dem Prinzip der Addition einzelner Sinusanteile, die jeweils im gewünschten Verhältnis zueinander gewichtet werden können. Dies ist sozusagen eine Umkehrung der Fourier-Analyse: Während bei letzterer die Sinusanteile aus einer vorgegebenen Wellenform isoliert werden, werden hier die einzelnen Anteile zum gewünschten Klang zusammengefügt. Dabei sollte für jeden einzelnen Sinusanteil eine eigene Hüllkurve definierbar sein, um komplexe, naturnahe Lautstärke- und Frequenzverläufe erzeugen zu können.

Dies macht bereits den Hauptnachteil der additiven Synthese klar: Für jede Oberwelle wird ein eigener Oszillator benötigt, dessen ADSR-Werte obendrein noch einzeln programmiert werden müssen. In analoger Technik ist so etwas heutzutage gar nicht mehr rentabel, aber auch ein digitaler Hochleistungs-Signalprozessor stößt hier schnell an seine Grenzen. In den meisten additiven Synthesizern werden deshalb nicht mehr als 32 Teiltöne pro Tongenerator erzeugt, was allerdings schon ein ziemlich sattes Klangbild ermöglicht. Dieses Verfahren bietet den direktesten Zusammenhang zwischen Parameter-Eingabe und Klangergebnis. Da die spektralen Verteilungen der Sounds direkt als Fourier-Koeffizienten angegeben werden, ist hier eine sehr präzise Klangkonstruktion möglich.

Subtraktive Synthese

Im Gegensatz zu additiven Synthese, werden hier keine Komponenten addiert, sondern von einer gegebenen Ausgangs-Wellenform subtrahiert, also ausgefiltert. Die zugrundeliegende Wellenform muß dabei möglichst obertonreich sein, wie z.B. ein Sägezahn oder ein Rauschen. Dieses Synthese-Verfahren ist vor allem für die Nachbildung von Klängen oder Instrumenten geeignet, die eine physikalische Entsprechung dieser Methode bilden: Zum Beispiel werden von einer Trompete lediglich gewisse Teile aus dem Geräusch der Lippen isoliert und entsprechend verstärkt, was ja gewissermaßen ein subtraktiver Vorgang ist. Auch die menschliche Stimme arbeitet nach diesem Prinzip, da die Artikulationsorgane lediglich bestimmte Obertöne (Formanten) hervorheben und andere dämpfen, um die verschiedenen Vokale zu erzeugen.

Ein besonderer Vorteil der subtraktiven Synthese liegt darin, ebenso geradzahlige wie unharmonische, diskrete wie kontinuierliche Oberwellenverteilungen erzeugen zu können. Als Filter werden heutzutage hauptsächlich Digitalfilter eingesetzt, da diese den Analogen in vieler Hinsicht überlegen sind. Auch die Generierung des Basissignals wird meist von digitalen Bausteinen erledigt, womit man eine recht gute Kontrolle über den ursprünglichen und damit auch den resultierenden Obertongehalt gewinnt.

Nichtlineare (Distortion-)Synthese

Dieses Synthese-Verfahren benutzt Verzerrungen, um aus einer vorgegebenen Wellenform (meistens ein einfacher Sinus) eine komplexere, obertonreichere zu machen. Dies wird mit Hilfe einer nichtlinearen Übertragungsfunktion bewerkstelligt, die bestimmten Eingangsamplituden anderen Ausgangsamplituden zuweist, und dies eben mit einer nichtlinearen Kennlinie. Abb. 5.5 zeigt die Behandlung eines Sinussignals mit verschiedenen Kennlinien.

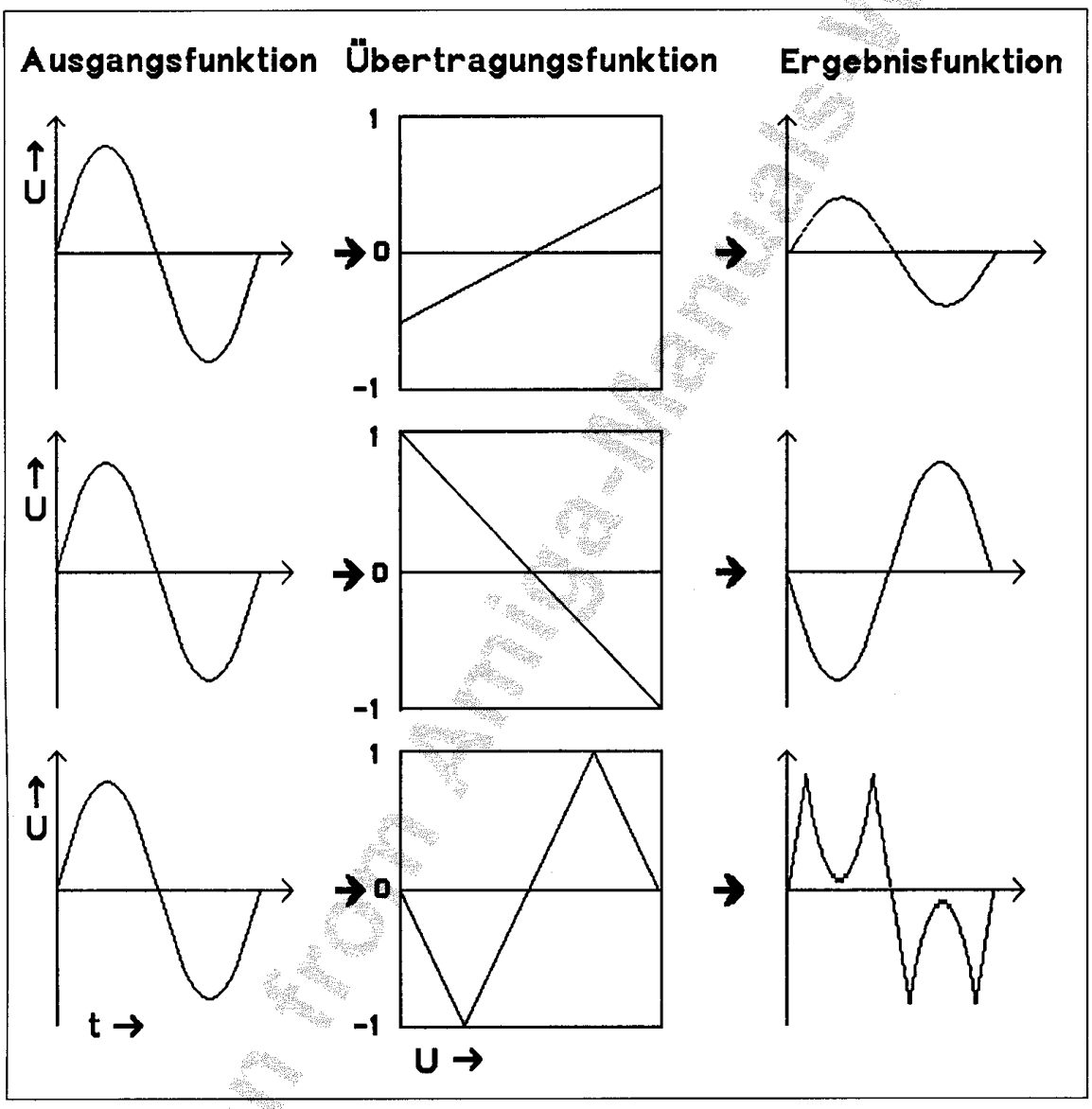


Abb. 5.5: Verzerrung durch nichtlineare Übertragungsfunktionen (Aus: Richard Aicher, Das MIDI-Praxisbuch, 1988, Signum Medien Verlag)

Der größte Vorteil dieses Verfahrens ist, daß die hardwaremäßige Realisierung sehr einfach, bzw. der Rechenzeitbedarf sehr gering ist: Jedem Sample des Klang-Datenstromes braucht nur ein

bestimmter Wert zugewiesen werden, der aus einer Tabelle gelesen wird. Dies ist z.B beim 68000-Prozessor innerhalb von 24 Buszyklen erledigt! Ein gewichtiger Nachteil ergibt sich jedoch aus dem sehr komplizierten Verhältnis zwischen Eingangsignal, Kennlinie und Ergebnissignal. Die Kennlinie muß, zumindest wenn man einzelne Frequenz-Anteile berechnen will, mit Hilfe sog. Tschebyscheff-Polynome konstruiert werden. Dabei wird die gewünschte Gewichtung der Obertöne in die jeweiligen Polynome eingesetzt und dann die Verzerrungsfunktion aus den einzelnen Polynomen zusammenaddiert. Diese Berechnungen werden jedoch in den meisten Geräten, die mit diesem Syntheseprinzip arbeiten, durch den internen Computer hinreichend unterstützt, so daß der Benutzer nur noch die gewünschten Oberwellenverteilungen eingeben muß. Deshalb bietet dieses Verfahren nicht nur eine einfache und überschaubare Möglichkeit zur Klangsynthese, es ist darüber hinaus auch noch sehr effizient und erlaubt die Realisierung sehr vieler Soundgeneratoren mit einer einzigen digitalen Recheneinheit.

Phase Distortion-Synthese

Einige sehr beliebte Synthesizer, wie z.B. der gute alte Yamaha DX-7, wenden gewissermaßen eine Kombination aus FM- und Distortion-Synthese an. Sie beruht auf dem Prinzip der nichtlinearen Änderung der Phasenlage einer Wellenform, weshalb sie Phase-Distortion-Synthese genannt wird. Hier wird zwar meist auch nur eine Sinustabelle ausgelesen, die Adresse in der Tabelle wird aber nicht nacheinander um einen Wert erhöht (wie bei der einfachen Sampling- oder PCM-Synthese), sondern wird durch eine nichtlineare Funktion bestimmt *(siehe Abb. 5.2)*.

Auch hier ist der Hardware- bzw. Rechenzeitaufwand relativ gering, da nur aus einer Modulations-Tabelle der richtige Wert in der Sinustabelle bestimmt wird, der als nächstes ausgegeben werden muß. Der Zusammenhang zwischen Modulations-Tabelle und Klangergebnis ist allerdings ziemlich kompliziert, so daß der Anwender nur mit Hilfe einiger praktischer Erfahrung mit dieser Technik umgehen kann.

Frequenzmodulations-(FM-)Synthese

Dieses Syntheseverfahren bedient sich des aus der Nachrichten- und speziell Funktechnik bekannten Prinzips der Frequenzmodulation. Hier wird nicht die Phasenlage, sondern die die Geschwindigkeit des Phasendurchlaufs beeinflußt. Es wird also eine Frequenz (auch hier meist nur eine einzelne Sinusschwingung) periodisch in ihrer Tonhöhe verändert, was zu sog. Intermodulationsverzerrungen führt. Bemerkenswerterweise entsteht dabei kein kontinuierliches Spektrum, sondern eine diskrete Verteilung der Oberwellen. Die Originalfrequenz wird hierdurch nämlich nicht einfach über einen bestimmten Bereich verteilt (wie man annehmen könnte), sondern es entstehen genau definierte Seitenbänder im Abstand von fm, der Modulationsfrequenz. Das Verhältnis zwischen den Seitenbändern und der Originalfrequenz wird dabei von der Amplitude der Modulation bestimmt.

In der (digitalen) Praxis wird eine Frequenzmodulation auf folgende Weise realisiert: Als Ausgangsfunktion wird eine Sinustabelle zugrundegelegt. Die Auslesegeschwindigkeit dieser Tabelle (Sampling-Rate) ist jedoch nicht konstant, sondern wird von einer weiteren Tabelle, der Modulationstabelle, bestimmt. Auch dieses Verfahren ist sehr leicht auf einem digitalen Prozessor realisierbar.

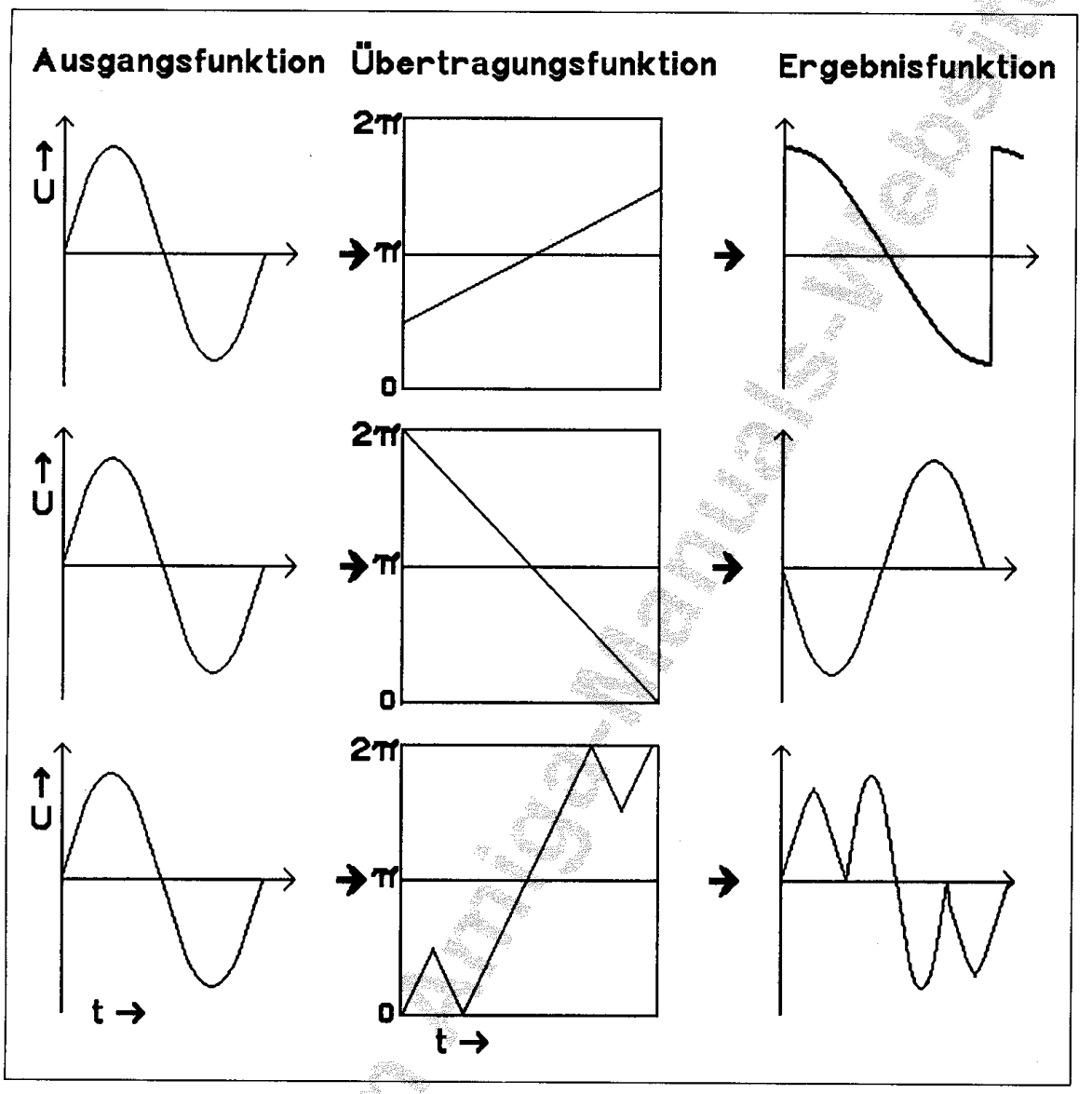


Abb. 5.6: Verzerrung durch Phasenmodulation (Aus Richard Aicher, Das MIDI-Praxisbuch, 1988, Signum Medien Verlag)

5.3.2 MIDI-kompatible Synthesizer und andere Geräte

Bei der explosionsartigen Vermehrung der MIDI-Geräte auf dem Markt ist es mit der Zeit immer schwerer geworden, einigermaßen die Übersicht zu bewahren und somit sinnvolle Kaufentscheidungen treffen zu können. Durch die enorm schnellen Fortschritte in der Integration von elektronischen Bauteilen ermöglicht, kommen ständig immer billigere und leistungsfähigere Synthesizer auf den Markt, welche meist die Sensationen vom Vorjahr schon wieder veraltet erscheinen lassen. Die Möglichkeiten dieser Geräte sind heutzutage oft so spezialisiert und komplex, daß der normale Anwender mit der Aufgabe, die genau ideale Lösung für seine

Ansprüche und Wünsche zu finden, in der Regel überfordert ist. Sind die eigenen Vorstellungen auch noch sehr vage und ungenau, hilft eigentlich nur ein detailliertes Einführungsgespräch mit einem Fachberater im nächsten Musikalien-Shop; und was den Klang betrifft, sollte man auf jeden Fall zuerst einige Zeit an einem Vorführgerät verbringen, bevor man sich zum Kauf entschließt. In der Regel nimmt die anfängliche Begeisterung für den beeindruckenden Sound eines Gerätes sehr schnell ab, wenn man einige Zeit damit gearbeitet hat und zu rasch an seine klanglichen Grenzen stößt. Meistens besitzt ein Synthesizer jedoch schon eine große Anzahl von guten Voreinstellungen, mit denen man sich ein Bild von den Fähigkeiten des Geräts machen kann. Also unbedingt alles vorher ausprobieren! Erst wenn man das derzeitige Angebot und die Fähigkeiten der Modelle ungefähr kennt, bekommt man eine Vorstellung über die Realisierungsmöglichkeiten seiner musikalischen Pläne.

Dieser Abschnitt soll eine kleine Übersicht über die meistverwendeten Geräte der unteren bis mittleren Preisklasse geben. Hier wird natürlich kein Anspruch auf Vollständigkeit gestellt, dies wäre wegen der Vielzahl und der hohen Innovationsgeschwindigkeit der Geräte auch gar nicht möglich. Es soll vielmehr ein kleiner Einblick in die Arbeitsweise, die Möglichkeiten und Bedienung heutiger Synthesizer und Sound-Maschinen geboten werden, um auch dem Neuling oder dem reinen Computer-User eine Orientierungshilfe zu geben.

Keyboards

Die MIDI-Schnittstelle wurde ursprünglich für Keyboards entworfen, also für Synthesizer mit einer Klaviatur; dies wird bei so spezifischen Parametern wie After-Touch oder Pitch-Weehl deutlich. Auch heute wird das Keyboard noch als das Standard-MIDI-Instrument angesehen. Hinsichtlich der Qualität der Ausführung und der klanglichen Möglichkeiten gibt es hier natürlich gewaltige Unterschiede: Zum Beispiel sind die meisten Geräte der unteren und mittleren Preisklasse mit einer sehr leichten Plastik-Tastatur ausgestattet, die nur wenig Spielgefühl vermittelt und einem Musiker, der Klaviertastaturen gewöhnt ist, einiges an Umstellung abverlangt. Dies kann für schnelle Keyboard-Solos ganz nützlich sein, ein realistischeres Anschlagsverhalten erzielt man jedoch mit gewichteten Tasten, die so die Hammermechanik eines Klaviers simulieren und damit einem »natürlichen« Instrument ähnlicher sind. Vor allem sollte ein Keyboard sämtliche Bedienelemente aufweisen, die hier üblich sind, d.h. einen Pitch-Bender, ein Modulationsrad und möglichst auch einen oder mehrere »analoge« Schieberegler für schnelle Dateneingabe.

Eine weitere wesentliche Eigenschaft sind die Möglichkeiten des Keyboard-Splitting, der Aufteilung der Tastatur in bestimmte Sektionen, die möglichst viele Parameter des internen Klangerzeugers beliebig beeinflussen können sollten. Es sollte z.B. kein Problem sein, dem unteren Drittel der Klaviatur den Sound A, dem mittleren Sound A und B gemischt und dem oberen Drittel Sound A, B und C zuzuordnen, wobei C nur unter bestimmten Bedingungen erklingt, wie eine hohe oder niedrige Anschlagsdynamik etc. Für den Liveeinsatz ist es obendrein sehr nützlich, wenn diese einzelnen Keyboard-Splits auf verschiedenen MIDI-Kanälen senden können, um einzelne Slaves gezielt ansteuern zu können (bei der Verwendung eines Sequencers ist das meist nicht nötig). Auf jeden Fall sollte aber zumindest ein gemeinsamer Sendekanal frei eingestellt werden können, auch eine Local-on/off- Funktion (Trennung der Klangeinheit von der Klaviatur) sollte unbedingt vorhanden sein.

Synthesizer mit PCM-Synthese

Kawai K1II

Äußerlich macht der K1 den Eindruck eines durchschnittlichen, soliden Musikwerkzeugs. Die Klaviatur umfaßt 61 Tasten, die zwar einen Aftertouch-, aber keinen Polyphonic-Key-Pressure-Wert liefern. Die Tasten sind in Plastik mit einem sehr leichten Anschlag ausgeführt, was das typische Keyboard-Spielgefühl vermittelt. Außer dem Modulationsrad und dem Pitch-Bender, der eine für meinen Geschmack etwas zu hohe Rückstellkraft aufweist, verfügt der K1II noch über einen Volume-Regler und einen Joystick, über den Parametereingaben vorgenommen werden können. Die von der Klaviatur abgegebenen Anschlags-Informationen sind leider ein wenig undifferenziert, was aber durch die Einstellung der entsprechenden Parameter wettgemacht werden kann.

Der Kawai K1II ist ein typischer Vertreter der reinen PCM-Synthesizer, er erzeugt seine Sounds also ausschließlich über gespeicherte PCM-Wellendaten. Hier stehen 204 VM- (kurze, berechnete Wellenformen, die ein konstantes Spektralbild liefern) und 52 PCM-Waveforms (längere Samples, wie Drums oder Stimme), also insgesamt 256 verschiedene Wellenformen zur Verfügung. Der K1II besitzt 32 unabhängige Tongeneratoren, die jeweils eine bestimmte Wellenform wiedergeben können. Die 32 Tongeneratoren können jedoch nicht einzeln gespielt werden, sondern werden zu Gruppen von jeweils zwei oder vier zusammengefaßt. Daraus ergibt sich eine Anzahl von entweder 16 (2 Generatoren) oder 8 (4 Generatoren) unabhängig spielbarer Stimmen. Jeder der Tongeneratoren kann mit eigenen Parametern für Hüllkurve, Vibrato und vieles mehr belegt werden, was bereits sehr komplexe Klangverläufe ermöglicht.

Darüber hinaus beinhaltet der K1II gegenüber seinem Vorgängermodell, dem K1, noch eine separate Drum-Sektion mit 32 PCM-Drumsounds sowie eine Effekt-Abteilung mit 16 digitalen Hall- und Delayprogrammen, die den Klang dieses Synthesizers merklich aufwerten.

Diese Form der Zusammenfassung der Klangquellen wurde hier »Single-Programm« getauft. Ein solches Single-Programm besteht aus einem Satz von Parametern, die die Art und Weise des Zusammenwirkens der einzelnen Generatoren steuern. Diese teilen sich ein in Generator-spezifische Parameter, wie z.B die verwendete Wellenform, und solche, die für alle Klangquellen eines Single-Programms gelten, wie das Vibrato.

Letztere werden wegen ihrer Wirkung auf alle Generatoren auch Common-Parameter genannt. Unter ihnen sind auch zwei Funktionen zur dynamischen Modulation der Tonhöhe vorgesehen, das Vibrato und der sog. Auto-Bend, der ein »Gleiten« der Frequenz von einem höheren oder tieferen Ton auf die eigentliche Endtonhöhe ermöglicht. Diese beiden Funktionen lassen sich auf vielerlei Art beeinflussen und verändern: Der Auto-Bend wird durch die Differenz zwischen Anfangs- und Endtonhöhe sowie durch die Zeit bestimmt, die vom Tastendruck bis zum Erreichen der endgültigen Tonhöhe vergeht. Hierdurch läßt sich u.A. ein »schmatzender« Sound erzielen, aber auch langsame, gedehnte Frequenzänderungen. Er kann außerdem von der Anschlagstärke sowie von der Lage des gespielten Tones beeinflußt werden. Man kann z.B. die Auto-Bend-Dauer im unteren Tastaturbereich lang wählen und in den oberen Lagen kurz, wobei man einen sprunghaften oder allmählichen Übergang zwischen den Lagen wählen kann. Des weiteren läßt sich auch der direkte Einfluß des Tastendrucks (Aftertouch) und des Pitch-Benders auf die

Tonhöhe programmieren. Das Vibrato hingegen läßt sich in seiner Tiefe, Geschwindigkeit und in der LFO-Wellenform manipulieren, es kann also Sinus-, Rechteck-, Sägezahn- oder die Form einer Zufallsfunktion haben. Auch läßt es sich vom Tastendruck in seiner Tiefe und vom Modulationsrad entweder in Tiefe oder Geschwindigkeit beeinflussen.

Die zweite Gruppe von Parametern wird Freq-Gruppe genannt. Hier befinden sich neben den Ein-/Aus-Schaltern für Vibrato und Auto-Bend noch die Paramater zur Tonhöhenverstellung, Freq Coarse und Freq Fine. Mit diesen beiden Reglern lassen sich die einzelnen Generatoren beliebig gegeneinander verstimmen, um gute Chorus- und Stereoeffekte zu erzielen. Diese und alle folgenden Gruppen beziehen sich jeweils auf einen einzelnen Tongenerator, und können für jeden von ihnen getrennt eingestellt werden.

Die dritte Gruppe wird Wave-Gruppe genannt und umfaßt die Wahl der Wellenform für den jeweiligen Generator, die Angabe der Kombination der Amplituden-Modulation und eine Copy-Funktion, mit der die Daten eines Single-Programms an ein anderes kopiert werden können. Bei der Amplituden-Modulation regelt ein Generator die Lautstärke eines beliebigen anderen, womit Verzerrungseffekte erzeugbar sind, die die Klangmöglichkeiten des K1II nicht unwesentlich erweitern.

Die vierte Gruppe wird ENV-Gruppe genannt, da sie die sämtliche Hüllkurven-Parameter enthält. Natürlich können hier alle Standard-Hüllkurvenwerte, Attack, Sustain, Decay und Release programmiert werden, sowie die Gesamtlautstärke und ein »Delay« genannter Parameter zur Verzögerung des gesamten Hüllkurven-Ablaufs. Darüber hinaus kann der Lautstärkeverlauf noch von der Anschlagsdynamik, dem Aftertouch und der Tonlage beeinflußt werden.

Mit der Konstruktion von Single-Programmen sind die Möglichkeiten des K1II aber bei weitem noch nicht ausgeschöpft: Seine wahren Kapazitäten offenbart der K1II erst bei der Kombination von mehreren Single-Programmen, die gleichzeitig erklingen. Dabei können die einzelnen Singles von vielen verschiedenen Bedingungen, wie der Tonlage (Keyboard-Splitting!), der Anschlagsdynamik etc. manipuliert werden. Auch können hier den Single-Programmen bestimmte MIDI-Kanäle zugewiesen werden, was den K1II zu einem vollen Multi-Mode-Synthesizer macht. Eine solche Zusammenfassung von Single-Programmen wird Multi-Programm genannt. Ein solches Multi-Programm besteht aus bis zu acht Single-Programmen, und definiert die folgenden Parameter:

- die Gesamtlautstärke des Multi-Programms,
- die Single-Programme, die in ihm vorkommen,
- den Tastaturbereich, auf den ein Single-Programm reagieren soll,
- die Anzahl der Stimmen, die einem Single zur Verfügung stehen (oder variabel),
- den Bereich der Anschlagsdynamik, auf den ein Single-Programm reagieren soll,
- die Quelle des Spielbefehls: Keyboard, MIDI oder beides zusammen (d.h. ein Local on/off für jede einzelne Stimme),
- die MIDI-Kanäle, über den die jeweiligen Single-Programme gespielt werden sollen.

Mit allen diesen Fähigkeiten ist der K1II bereits in der Lage, schon sehr komplexe, vielschichtige Klangbilder zu synthetisieren. Es ist jedoch auch zu sagen, daß die von diesem Gerät erzeugten Sounds zwar durchweg voll und beeindruckend, aber auch sehr künstlich klingen. Der K1II ist

deshalb als ein sehr leistungsfähiger und flexibler Synthesizer zu bezeichnen. Er kann auch bedingt als Masterkeyboard eingesetzt werden, obgleich er nur auf einem einzigen Kanal zu senden in der Lage ist. Er sollte in einem MIDI-System jedoch auf jeden Fall durch noch ein Instrument mit einem anderen Syntheseverfahren ergänzt werden.

Synthesizer mit FM-Synthese

Yamaha V50

Der Yamaha V50 ist mit Fug und Recht als Profi-Gerät zu bezeichnen. Er fällt unter den hier besprochenen Geräten insofern etwas aus der Reihe, als daß er neben dem eigentlichen FM-Synthesizer noch eine vollwertige PCM-Rhythmuseinheit und einen Sequencer besitzt. Dies schlägt sich leider auch auf den Preis nieder: Mit ca. 3000 DM ist er für den Hobby-Musiker bereits eine größere Anschaffung, die wohlüberlegt sein will.

Dafür erhält man allerdings auch ein hervorragend ausgetattetes Gerät: leichtgängige Pitchbend- und Modulationräder sowie ein Lautstärke- und ein Data-Entry-Regler ermöglichen eine leichte Kontrolle; die gut verarbeitete Tastatur ist leicht spielbar und überträgt sehr fein abgestufte Dynamikwerte. Wie auch in dieser Preisklasse noch üblich, liefert die Tastatur zwar Aftertouch-, aber keine Polyphonic-Key-Pressure-Informationen. Die ungeheure Vielzahl der Funktionen des V50 ist mit dem ergonomisch gut gestalteten Bedienfeld relativ leicht zu bewältigen. Das beleuchtete 2x40-Zeichen-Display mit den zugehörigen acht Funktionstasten erlaubt stets eine sehr komfortable Parametereingabe. Neben einem RAM-Card-Slot ist als Speichermöglichkeit für Sound- und Sequenzerdaten sogar ein 3,5-Zoll-Diskettenlaufwerk eingebaut.

Neben den FM-Tongeneratoren des Synthesizers besitzt der V50 wie gesagt eine Drum-Machine, der ein eigener (achtstimmiger) PCM-Tongenerator mit 61 verschiedenen Sounds zugeordnet ist. Die Drum-Machine bildet selbst einen Pattern-orientierten Sequencer, der mit dem eigentlichen Sequencer-Teil sehr gut zusammenarbeitet. Selbstverständlich können sämtliche Daten wie Patterns und Songs auf RAM Card oder Diskette gespeichert werden.

Wie bei fast allen modernen Synthesizern, ist die Steuerung der Klangerzeugung in zwei Ebenen aufgeteilt: eine Ebene, in der die Soundeinstellungen der Tongeneratoren definiert werden, und eine zweite für die Zusammenfassung dieser Sounds zu einem Betriebsmodus. Diese Bedienungsebenen werden beim V50 Single- und Performance-Mode genannt. Da die Parameter des Performance-Modes im wesentlichen den der anderen hier besprochenen Geräten entsprechen (vgl. Multi-Mode beim Kawai K1II), soll im folgenden hauptsächlich auf den Single-Mode, also die Klangerzeugung selbst eingegangen werden.

Den Schlüssel zu der beachtlichen Klangvielfalt des V50 bildet die FM-Einheit. Um deren Funktionsweise zu verstehen, bedarf es in der Praxis schon einer recht guten Kenntnis der Theorie der FM-Synthese; diesem Umstand wird im Benutzerhandbuch allerdings genügend Rechnung getragen. Eine Tongenerator-Einheit, hier wieder Voice genannt, besteht aus vier einzelnen Oszillatoren, die auf zwei Arten miteinander verknüpft werden können: durch bloße Addition, also Mischung der beiden Ausgangssignale und indem der eine die Frequenz des anderen moduliert. Bei dieser Verknüpfung entsteht, wie oben in diesem Kapitel erwähnt, ein Spektrum mit Seitenbändern im Abstand der Frequenz des Modulators. Die vier Oszillatoren, auch

»Operatoren« genannt, können allerdings nur zu insgesamt acht verschiedenen Konfigurationen zusammengeschaltet werden. Operator Nummer 4 hat dabei eine Sonderstellung: Sein Ausgangssignal kann auf seinen Eingang rückgekoppelt werden.

Natürlich steht für die Erzeugung einer möglichst großen Vielzahl von Klängen auch eine ausgeklügelte Steuerung der Parameter zur Verfügung. Die wichtigsten Parameter sind hier:

- Algorithm, Feedback; wählt die Schaltkonfiguration der vier Operatoren sowie das Rückkopplungsverhältnis des 4. Operators
- LFO wave; Wellenform, die der LFO erzeugt
- LFO speed; Frequenz des LFO
- LFO delay; Verzögerung, mit der der LFO zu modulieren beginnt
- LFO Pitch Modulation Depth; Modulationstiefe des Vibratos
- LFO Amplitude Modulation Depth; Modulationstiefe des Tremolos
- Envelope Sensitivity; Einfluß des Hüllkurven-Generators auf die Frequenz
- Oscillator Mode; hier kann zwischen einer Klaviatur-abhängigen und einer fest vorgegebenen Frequenz für einen Operator gewählt werden
- Oscillator Coarse; dies ist der Faktor, mit dem die Tonhöhe für einen Operator multipliziert wird. Bei einem Faktor von 1 liegt zwischen zwei Klaviaturtasten ein Tonintervall von einem Halbton. Dies kann durch den Coarse-Faktor geändert werden
- Oscillator Fine; Feinregelung für das Frequenzverhältnis
- Oscillator Wave; Wellenform des jeweiligen Operators
- Hüllkurvengenerator mit Attack, Decay, Sustain, Release für jeden Operator
- Portamento Time; regelt die Dauer des Portamentos

Allgemein läßt sich sagen, daß der V50 ein sehr gutes Werkzeug sowohl für das kreative Arbeiten (musikalischer Notizblock) als auch für die Erzeugung von vornehmlich künstlich wirkenden Klängen bildet. Sein Preis erscheint unter Berücksichtigung seines hohen Leistungniveaus und seiner Vielseitigkeit durchaus gerechtfertigt.

Schlagzeugmaschinen und Drum-Sequencer

Diese Sorte von Geräten ist speziell für die Wiedergabe von Schlagzeug- und Percussionklängen konzipiert. Sie enthalten meist einen eigenen Sequencer, also eine Einheit zur Speicherung und Ausgabe von Rhythmen oder ganzen Songs, womit der Haupt-Sequencer des Systems wirksam entlastet werden kann. Um eine möglichst hohe Speicherausbeute zu gewährleisten, sind Schlagzeugmaschinen meist als sog. Pattern-Sequencer konzipiert, d.h. die gespeicherten Songs setzen sich aus einzelnen Teilen, den Patterns, zusammen. Diese können beliebig hintereinandergereiht und miteinander kombiniert werden, wobei ein einzelner Pattern nur einmal eingegeben werden muß und deshalb auch nur einmal im Speicher steht. Im Prinzip ist dieses Verfahren für so gut wie alle Zwecke ausreichend, obwohl man sich teilweise davor hüten muß, durch die bequeme Wiederholung von einigen Patterns eine unschöne Monotonität zu erzeugen.

Eine Schlagzeugmaschine kann zwar auch andere angeschlossene Geräte steuern und synchronisieren, wird aber in Verbindung mit einem Master-Sequencer (wir haben ja den Amiga) meist als reiner Slave benutzt, der vom Master gesteuert wird. Selbstverständlich sollte er auch andere

Slaves auf einem beliebigen Kanal ansteuern sowie seine internen Daten über eine Exclusive-Message an den Master weitergeben können. Die Vielfalt und Qualität der erzeugbaren Sounds hängt natürlich im wesentlichen von der Preisklasse ab, durch die Möglichkeiten der PCM-Synthese sind hier jedoch schon äußerst günstige Geräte auf dem Markt zu haben.

Yamaha RX-8

Die Yamaha RX-8-Schlagzeugmaschine ist im Grunde genommen ein PCM-Synthesizer mit einem integrierten Pattern-Sequencer. Neben den (leider nicht anschlagsdynamischen) Drumpad-Tasten aus Gummi, mit denen sowohl live gespielt als auch ein Rhythmus-Pattern eingegeben werden kann, verfügt der RX-8 noch über die Sequenzer-typischen Start/Stopp-Knöpfe, eine ganze Reihe von Dateneingabetasten sowie einen Volume- und Temporegler. Außerdem existieren zwei Tasten für Tonhöhenänderung und Emphase (Lautstärkenanhebung) der gerade gespielten Drums. Die voreingestellte Zuordnungen der einzelnen Drum-Tasten ist Bassdrum, Snaredrum, vier Tom-Toms, offenes und niedergetretenes HiHat, ein Ride- und ein Crash-Becken, sowie zwei Acoustic Bass-Sounds. Diese Zuordnung läßt sich natürlich beliebig auf die 43 verfügbaren PCM-Drums umverteilen.

Wie alle Pattern-orientierten Sequencer, so kennt auch der RX-8 zwei Bearbeitungsmodi, den Pattern-Modus und den Song-Modus. Im Pattern-Modus werden die Inhalte der einzelnen Patterns festgelegt, die nachher zu einem Song zusammengesetzt werden sollen. Hierbei stehen die folgenden Funktionen zur Verfügung:

- Select Pattern; hier wird das zu bearbeitende Pattern-Muster gewählt.
- Realtime Write; hier kann ein Pattern in Realzeit eingespielt werden. Dabei wird ein vorher bestimmtes Taktmaß (in 16tel angebbar) kontinuierlich wiederholt, und die in jeder einzelnen Wiederholung gespielten Elemente gespeichert. So kann man einen Pattern bequem aus einzelnen Drums erstellen, indem man z.B. beim ersten Durchlauf nur Bassdrum und Snare, beim zweiten die Toms, im dritten die Becken einspielt usw. Hierfür muß zuvor die Länge des Taktmaßes, die Quantize-Einstellung (also die Auflösung) und evtl. die Lautstärke des Metronom-Klicks eingestellt werden.
- Step Time Write; mit dieser Funktion kann ein Pattern bezüglich seiner einzelnen Stimmen ediert werden. Das Display übernimmt hier die Funktion eines Editor-Fensters, in dem die Aktivierungs-Events für eine einzelne Drum-Stimme dargestellt werden. Hier können die Schlagzeugnoten für jeden einzelnen Track an allen Stellen des Patterns gesetzt und gelöscht, und damit der gesamte Pattern bequem umgestaltet werden. Die Anzahl der Positionen im Takt, an denen Beats eingefügt werden können, richtet sich dabei nach dem Taktmaß und vor allem nach der Auflösung, die für den Pattern gewählt wurde.
- Clear Pattern; mit dieser Anweisung kann ein Pattern ganz gelöscht werden.
- Clear SD1; mit dieser Anweisung kann eine einzelne Stimme in einem Pattern gelöscht werden.
- Copy Pattern; damit kann ein Pattern auf einen andern kopiert werden, z.B. um in einfacher Weise Variationen eines bestimmten Patterns zu erzeugen.

- Clear All Patterns; diese Anweisung löscht alle 100 vorhandenen Patterns.
- Used Memory; zeigt den Grad der Füllung des für die Patterns reservierten Speicherplatzes an.

Der Song-Modus ist für die Konstruktion von ganzen Liedern aus einzelnen Patterns verantwortlich. Beim RX-8 können Songs aus bis zu 999 Patterns zusammengesetzt werden. Es können maximal 20 verschiedene Songs programmiert werden. Im Song-Modus stehen folgende Optionen zur Verfügung:

- Select Song; hier wird der zu bearbeitende Song ausgewählt.
- Edit Song; mit dieser Funktion gibt man die Reihenfolge der Patterns an, beim Start des Songs nacheinander gespielt werden. In jedem Eingabeschritt hat man die Wahl, entweder eine Pattern-Nummer oder eine Tempo- oder Lautstärkeänderung anzugeben; so kann man auf einfache Weise Accelerandos, Crescendos und Decrescendos erzeugen. Außerdem gibt es eine Möglichkeit, Wiederholungen bestimmter Teile des Songs zu programmieren: Es können Wiederholungszeichen gesetzt werden, die Anfang und Ende des Parts markieren und die Anzahl der Wiederholungen enthalten. Solche Repeats können auch geschachtelt sein, d.h. ein Wiederhol-Teil kann einen anderen enthalten, dieser wiederum einen anderen usw. Darüber hinaus stehen natürlich die obligatorischen Kopier-, Schneide- und Einfügefunktionen sowie eine Suchfunktion zur Verfügung.
- Clear Song; hiermit kann ein ganzer Song gelöscht werden.
- Copy Song; kopiert einen Song auf einen anderen, der alte wird dabei überschrieben.
- Clear All Songs; löscht alle Songs im Speicher.
- Used Memory; zeigt den Belegungsgrad des Song-Speichers an.

Eine der wichtigsten Eigenschaften einer Schlagzeugmaschine ist wohl ihre Fähigkeit, in einem MIDI-System korrekt und flexibel zu arbeiten. Auch für die Enstellung der verschiedenen MIDI-Parameter steht ein eigener Mode zur Verfügung. In ihm sind die folgenden Funktionen vorgesehen:

- Ch Message on/off; hier wird bestimmt, ob der RX-8-Kanalinformationen verarbeitet oder nicht, was gewissermaßen einem Omni-on/off-Schalter entspricht.
- Receive Ch; hier kann der Empfangskanal gewählt werden, auf dem alle Stimmen des RX-8 gespielt werden.
- Trans CH Assign; ermöglicht die Zuordnung jeder Stimme an einen eigenen Sendekanal.
- Note Assign; hier läßt sich bestimmen, ob die Drums in der angegebenen Tonhöhe gespielt werden (Pitch Note, Umfang 2 Oktaven) oder jede MIDI-Tonnummer ein anderes Instrument spielen soll (Voice Note). Die Zuordnung der Tonnummern an die Instrumente kann frei gewählt werden.
- Transmit Bulk; dieser Befehl sendet eine System Exclusive-Meldung auf den MIDI-Bus, die den gesamten Speicherinhalt z.B. an den Hauptsequencer zur Speicherung auf Diskette übertragen kann.

- Receive Bulk; empfängt einen Datensatz vom MIDI-Bus.
- Echo Back; schaltet die interne Echofunktion (Wiederausgeben der über MIDI IN hereinkommenden Daten) ein und aus.

Zusätzlich können über bestimmte Nebenfunktionen, die aus allen der oben aufgeführten Modi augerufen werden können, die Lautstärke, Tonhöhe, Stereoverteilung und der Detune-Effekt manipuliert werden. Auch kann die Tastenzuordnung beliebig verändert werden.

Zusammenfassend läßt sich sagen, daß der RX-8 sowohl ein recht leistungsfähiger Sequencer ist als auch eine gute Klangerzeugung besitzt, was in zu einem geeigneten Partner für einen Master-Sequencer macht. Auch der Preis ist für diese Leistung durchaus im Rahmen, so daß er für die Realisierung eines Homerecording-Studios sehr interessant ist.

Synthesizer-Slaves (Expander)

Am weitaus schnellsten unter den MIDI-Geräten hat sich das Angebot der reinen Synthesizer-Slaves vergrößert, der sogenannten Expander. Sie besitzen keine Klaviatur oder einen anderen Controller, sondern werden ausschließlich über MIDI angesteuert. Ein solcher Slave besteht in der Regel nur aus einem Steuercomputer, der die Bedienelemente und die Schnittstelle versorgt, und einigen Signalprozessoren und D/A-Wandlern. Deshalb können hier sehr preiswerte Geräte angeboten werden, die schon eine erstaunliche Leistung bieten. Vor allem für einen Sequencer ist ein Expander der ideale Partner, da sie bezüglich der Stimmen- und Instrumentverteilung meist sehr flexibel sind.

Die wichtigste Eigenschafte eines Expanders ist zuerst einmal seine Fähigkeit, möglichst alle MIDI-Botschaften empfangen und verarbeiten zu können. Die MIDI-Kanäle müssen unabhängig programmierbaren Sound-Gruppen zuzuordnen sein; auch sollte man die einzelnen Stimmen vollkommen dynamisch auf die Gruppen verteilen können (MULTI-Mode). Ein weiterer Faktor ist der Bedienungskomfort bei der Dateneingabe: Wenn man die Möglichkeiten eines Gerätes wirklich voll ausschöpfen will, möchte man sich nicht mit einem winzigen Display und ewig langen Menüs herumärgern müssen. Die Bedienpanels der meisten Expander sind jedoch meist ziemlich dürftig ausgestattet, oft sind alle Funktionen nur über Tastknöpfe anwählbar, was bei häufigem Ändern von Parametern bald zu Qual wird.

Wichtig ist auch das Vorhandensein mehrerer getrennt schaltbarer Ausgänge, auf die die einzelnen Soundgruppen verteilt werden können. Spätestens wenn man einige Effektgeräte einsetzt, wie Hall oder Echo, möchte man einige Stimmen getrennt mit dem Effekt versehen, und die anderen unverändert lassen. Deshalb sollte sowohl eine Umschaltmöglichkeit zwischen den Ausgängen, als auch eine Stereo-Verteilung (Panning) vorgesehen sein, um dem Klang eine Ortung innerhalb des Stereobilds zu verleihen.

Roland U-220

Der Roland U-220-Expander ist ein weiterer Vertreter der Familie der PCM-Synthesizer. Er erzeugt seine Klänge jedoch nicht durch durch berechnete Wellenformen, sondern hauptsächlich durch akustisch erzeugte und dann gesamplete Instrumente. Der U-220 besitzt einige sehr brauchbare und in ihrer Art unübertroffene Sounds, die durch ihre große Fülle und Authentizität

begeistern. Hierbei wurde ein echtes Multi-Sampling vorgenommen, d.h. bei vielen Instrumenten steht über fast den gesamten Klaviaturbereich ein Sample pro Quart zur Verfügung. Bei den meisten Klängen ist daher an den Grenzen zwischen den einzelnen Samples kaum ein Verfremdungseffekt zu bemerken. Das braucht natürlich recht viel Speicher: immerhin 24 Megabit Sampledaten stehen in den ROMs des U-220 zum Abruf bereit. One-Shot-Wellenformen, wie die diversen Pianos, haben eine Länge von bis zu zehn Sekunden, geloopte Formen haben immerhin, je nach Tonhöhe, ein Wiederholungsintervall bis zu drei Sekunden. Letztere klingen durch ihre doch recht große Länge und ihre optimalen Loop-Points sehr realistisch und voll, ohne den bei geloopten Samples gefürchteten Monotonitäts-Effekt aufkommen zu lassen.

Der U-220 ist, wie die meisten Slaves, als 19"»-Gehäuse« ausgeführt. Die Frontplatte weist neben dem Volume-Regler, der Kopfhörerbuchse, dem (kärglichen) 2x16-Zeichen-Display und den (noch kärglicheren) zwölf Eingabetasten noch immerhin zwei Cartridge-Slots auf, über die seine Klangvielfalt noch wesentlich erweitert werden kann. Auf der Rückseite des Geräts befinden sich außer den MIDI-Anschlüssen noch sechs Ausgänge, zwei für normalen Stereo-Betrieb, und weitere vier für die getrennte Kanalausgabe an Mischpult oder Effektgeräte. Die Zuordnung der Sound-Einstellungen (oder Parts, wie sie hier heißen) an die Ausgänge kann jedoch nicht ganz frei gewählt werden, sondern man muß sich mit ca. 50 verschiedenen Zuordnungen zufriedengeben. Diese sog. Part-Assignments werden aus einer Tabelle entnommen, die allerdings wohl jeder wünschbaren Kombination Rechnung trägt.

Die Bedienung des U-220 ist wegen des geringen Eingabekomforts und nicht zuletzt wegen der Vielzahl der Funktionen relativ komplex, weshalb man sich die mitgelieferte »Operation Map« am besten in Sichtweite des Geräts aufhängen sollte. Die Parameter-Eingabe ist aufgeteilt in den Setup-, den Utility- und den Patch-Mode. Im Setup-Mode werden die MIDI-Empfangsparameter, der Master-Tune und die Zuordnung der Program-Change-Befehle an die einzelnen Soundprogramme festgelegt. Der Utility-Mode ist für den Displaykontrast, die Exclusive-Datenübertragung und den Demo-Mode zuständig, in dem vier recht beeindruckende Songs abgespielt werden können. Im Patch Edit-Mode lassen sich die Soundeinstellungen regeln. Dieser ist wohl der interessanteste der Eingabe-Modes, da sich hier der Zugang zu den klanglichen Möglichkeiten und Variationen befindet.

Die sogenannten Patches bestimmen die Konfiguration des gesamten Geräts (vgl. Multi-Modes beim Kawai K1II etc.). Der Patch-Edit-Mode ist unterteilt in den Write-Mode, die Common- und die Part-Settings. Der Write-Mode erlaubt das Kopieren von einem Patch auf den anderen. Die Common-Settings enthalten den Namen des Patch, den Output-Modus, und die Anwendungsart des integrierten Effekt-Prozessors. Die Part-Settings bilden den wichtigsten Teil eines Patch: Hier werden die Eigenschaften der einzelnen Stimmen (Parts) angegeben. Der U-220 besitzt 30 Tongeneratoren, die entsprechend dynamisch an die einzelnen Parts verteilt werden können; die Zuordnung der Stimmen wird mit Hilfe der oben genannten Tabelle festgelegt. Insgesamt sechs verschiedene Parts sind vorgesehen, deren Klang individuell gestaltet werden kann. Die Parameter eines Parts teilen sich wiederum in die Basic, die Level, Pitch- und LFO-Parameter ein:

Basic

Tone Select; hier wird das Sample-Set aus den 128 möglichen Kombinationen gewählt.

Output Assign; bestimmt die Zuordnung der einzelnen Stimmen des Parts an die Ausgänge des U-220.
MIDI RCV Channel; MIDI-Empfangskanal des Parts.
Program Change; bestimmt, ob der Program-Change-Befehl auf diesen Part eine Wirkung haben soll oder nicht.
PGM – Tone-Map; bezeichnet die Map der Soundeinstellungen, die auf die einzelnen Programm-Change-Befehle gewählt wird (siehe Setup-Mode).
Key Range Lo + Hi; dies ist der Tonlagenbereich, auf den der Part reagieren soll.

Level (Amplitudenänderung)

Part Level; Lautstärke des Parts.
Velo Sens; bestimmt die Auswirkungen der Anschlagsdynamik auf den Lautstärkeverlauf.
Env Attack; Attack-Wert (Anstiegsdauer) des Sounds.
Env Release; Release-Wert des Sounds.
Ch Pressure Sens; bestimmt die Auswirkungen des Aftertouch (bzw. des Polyphonic Key-Pressure) auf den Lautstärkeverlauf.

Pitch (Tonhöhenvariation)

Shift Coarse; Hier kann man die Stimmung des Patch halbtonweise verändern.
Shift Fine; dient zur Feineinstellung der Stimmung.
Bend Range; bestimmt die Frequenzänderung durch den Pitch-Bender.
Detune Depth; Bei vielen Sounds steht auch eine sog. Detune-Option zur Verfügung: Hier kann eine Wellenform mit einer gewissen Verstimmung zwischen den beiden Stereo-Ausgängen gespielt werden, was einen teilweise sehr schönen Chorus- oder Flanging-Effekt bewirkt. Der Detune Depth-Parameter bestimmt die Differenz zwischen den beiden Tonhöhen.
Pressure Sens; bestimmt die Auswirkungen des Aftertouch (bzw. des Polyphonic Key-Pressure) auf die Tonhöhe.

LFO (Vibrato)

LFO Rate; Geschwindigkeit des Vibratos.
Auto Depth, Delay Time, Rise Time; Hier wird die Charakteristik des Vibratos bestimmt. Hierdurch wird sozusagen die Hüllkurve des LFO definiert, um dessen zeitliches Verhalten genau steuern zu können. »Delay« bezeichnet die Verzögerung vom Tastenanschlag bis zum Beginn des Vibratos, »Rise« die Anstiegsrate des Vibratos bis zum »Depth«- Wert, der dann für die gesamte Dauer des Tones gehalten wird.
Manual Depth, Rise Time; Hier kann man die Wirkung des Modulationsrads auf das Vibrato bestimmen.
Ch, Poly Pressure Sens; Die Auswirkungen von normalem After-Touch und Polyphonic Key Pressure können hier getrennt geregelt werden.

Aus der Sicht des Musikers läßt sich sagen, daß der U-220 eine echte Bereicherung des Klangmaschinen-Parks eines jeden Homerecording- oder Profistudios ist. Durch das beim U-220 angewandte Multisampling sind sehr warme, analogähnliche Klänge realisierbar, die für fast alle Stilrichtungen einsetzbar sind.

Multitrack- und Masterrecording 6

Ist man als gestandener Amiga-Anwender wirklich von dem Wunsch beseelt, seinen musikalischen Ideen eine eindrucksvolle reale Gestalt zu geben, so führt dies zwangsläufig in die Fangarme eines epidemieartig um sich greifenden Syndroms, des Homerecordings. Die enormen Preissenkungen auf dem Audio-Gerätemarkt haben außer der heute enormen Flexibilität des Semiprofessionellen und Amateurmusikers zu einem neuen Typus von Musikern geführt, dem allein vor sich hin programmierenden Sequenzer-Freak. Seit Anbeginn wird seitens der Industrie dem Trend zu dieser scheinbar lebensausfüllenden Beschäftigung ausgiebig Rechnung getragen; ein einzelner Mensch in seinem Wohnzimmer kann heutzutage Aufnahmen in einer Qualität herstellen, von der man vor zehn Jahren in den Demostudios nur träumen konnte.

Auch der Amiga ist als zentrales Steuerelement eines Homerecording-Systems bestens geeignet. Dabei muß man jedoch ehrlicherweise sagen, daß andere Computer wie der Atari ST oder Apple Macintosh hier (noch!) eindeutig die Nase vorn haben. In den USA kein Studio ohne Mac, in Europa keins ohne Atari, so könnte man die Lage schildern. Für den Amiga haben sich bisher nur einige der hier führenden Hersteller erwärmen können, deren Angebotspalette wird aber wohl den meisten Ansprüchen gerecht.

Hat man sich erst einmal den ersten Synthesizer zugelegt, ihm über die MIDI-Schnittstelle die ersten Töne entlockt, erste Sequenzen ein- und übereinandergespielt und so die Möglichkeiten dieses Instrumentariums nur entfernt erahnt, so folgen in aller Regel bald weitere Anschaffungen: Expander für die Erweiterung der Soundmöglichkeiten und der Stimmenanzahl, Effektgeräte zur Bearbeitung und Verfremdung von Klängen, und als unausweichliche Konsequenz eine Mehrspur-Bandmaschine sowie eine Mischvorrichtung. Dieses Kapitel soll dem geneigten Leser einen kleinen Einblick in die Anschaffung und den Aufbau eines computerorientierten Homerecording-Systems bieten; als finanzielle Obergrenze für die Grundausrüstung wurden hierbei 5 000 bis 10 000 Mark zugrundegelegt.

6.1 Effekte

Trotz des ohnehin schon beeindruckenden Klanges der meisten Instrumente ist es fast immer notwendig, gewisse Veränderungen und Verfremdungen vorzunehmen, um dem Sound mehr Originalität, Vielschichtigkeit und Raumeindruck zu verleihen. Speziell bei der Aufnahme von Vocals und akustischen Instrumenten setzt man z.B. so gut wie immer einen gewissen Halleffekt ein, was mehr Tiefe erzeugt. Aber auch ein Keyboarder will nicht, daß seine Instrumente platt und vordergründig klingen, und ein Heavy-Metal-Gitarrist wird schwerlich ohne eine fetzige Distorsion auskommen. So werden auch hierfür eine Vielzahl von Geräten angeboten, die Palette reicht vom 50-DM-Transistor-Fuzz über die anspruchsvolleren und solideren Bühnengeräte bis

zum mehrere tausend Mark teuren Multieffekt-Prozessor, ganz zu schweigen von der professionellen Ausrüstung eines großen Studios. Hier nur eine ganz kleine Auswahl dessen, was man einem Klang so alles antun kann:

Hall und Echo

Ganz wesentlich für den räumlichen Eindruck eines Klanges ist sein Gehalt an Reflexionsgeräuschen. Kurze Reflexionszeiten spiegeln einen kleinen, lange einen großen Raum vor. Eine große Rolle spielt hierbei auch, welchen Absorptionsgrad die Wände des Raums aufweisen, wie viele Reflexionen also hintereinander wahrgenommen werden. Ein leerer Betonkeller wird sehr viele Reflexionen produzieren, ein schwer möblierter Raum klingt eher trocken. Nimmt man klar trennbare, einzelne Reflexionen wahr, so spricht man von einem Echo. Eine solche scharfe Trennung wird auf natürliche Weise nur in sehr speziellen Räumen erzeugt, z.B. in einem langen Gang. Ein reiner Echo-Effekt wird deshalb nur in wenigen Spezialfällen verwendet. In analogen Zeiten wurde dieser Effekt meist mit Hilfe einer Bandschleife erreicht, auf der das Signal an einer Stelle aufgesprochen und dahinter von mehreren Köpfen abgelesen wurde. Heutzutage wird so etwas natürlich nur noch digital realisiert, was der Analogtechnik hinsichtlich Rauschverhalten, Wiedergabetreue und nicht zuletzt im Preis eindeutig überlegen ist.

In der Realität ist es jedoch wesentlich häufiger der Fall, daß ein Geräusch in einem Raum eine sehr große Anzahl von Reflexionen mit verschiedenen Rückkehrzeiten und Intensitäten verursacht. Werden so viele Reflexionen übereinandergelagert, daß die einzelnen Echos nicht mehr voneinander unterschieden werden können, sondern zu einem konstanten Nachklingen verschmelzen, so spricht man von einem Nachhall. In jedem Raum, der nicht eine sehr spezielle geometrische Form aufweist, wie z.B. eine Kuppel, wird dies der vorherrschende Typ von Reflexionen sein. Naturgemäß ist ein solcher Effekt wesentlich schwerer mit elektronischen Mitteln nachzuahmen als ein einfaches Echo. Früher behalf man sich zu diesem Zwecke mit einer einfachen Metallspirale, in die das Signal mit Hilfe von Bewegungswandlern eingespeist und innerhalb der Feder vielfach reflektiert wurde. Die heutigen digitalen Geräte sind diesen Konstruktionen im Klang natürlich haushoch überlegen, aber auch an diesen lassen sich zuweilen Unzulänglichkeiten feststellen: Vor allem bei älteren Versionen, die noch nicht mit einer so üppigen Prozessorleistung ausgestattet waren wie es heute üblich ist, gelingt die Verschmelzung der Reflexionen nur ungenügend; man hört teilweise die einzelnen Echos heraus. Ein anderer für diese Technik bezeichnende Effekt ist die typische digitale Kälte des Sounds, der bei bestimmten unglücklichen Einstellungen sehr metallisch, hart und synthetisch wird. Beide Phänomene lassen auf eine unzureichende Anpassung an die Aufgabe bei der Entwicklung schließen, was den Einsatzbereich eines Gerätes freilich stark einschränken kann.

Aufgrund der freien Programmierbarkeit moderner Digitalbausteine bieten die meisten heute angebotenen Geräte eine Vielzahl von Einstellungen, die verschiedene besonders brauchbare Kombinationen aus Hall und Echo darstellen. Die Anzahl der Effekte sowie vor allem die der frei bestimmbaren Parameter, d.h. wieviel Einfluß der Musiker auf einzelne Komponenten eines Effekts hat, wird hauptsächlich durch die Preisklasse bestimmt. Im allgemeinen verfügen solche Geräte über Echoperioden bis zu einer Sekunde und einer Nachhalldauer bis zu ca. 10 Sekunden. Die Gerätepalette reicht vom kleinen Tischgerät, z.B. dem Yamaha R100 mit 60 festen Einstel-

lungen, bis zum Quantec-Raumklangsimulator, der die akustischen Verhältnisse so ziemlich jeden Ortes auf dieser Welt nachzuahmen imstande ist.

Kompression und andere Lautstärke-Effekte

Wie schon mehrfach erwähnt, ist die Hüllkurve eines Klanges, also sein Lautstärkeverlauf, ebenfalls eine für den Höreindruck sehr entscheidende Eigenschaft. In der Audiotechnik sind natürlich eine Vielzahl von Arten denkbar, in der die Hüllkurve beeinflußt werden kann. Möchte man z.B. das Sustain eines konstant ausschwingenden Instrumentes verlängern, so muß die Verstärkung angehoben werden, wenn der Eingangspegel unter einen bestimmten Wert absinkt, und ggf. nach unten korrigiert, wenn der Pegel darüberliegt. Das Prinzip einer solchen »Verengung« des Dynamikbereiches bezeichnet man auch als Kompression, welches auch für die Rauschverminderung bei Bandaufzeichnungen Verwendung findet.

Es existieren jedoch noch eine ganze Menge anderer Möglichkeiten: Zum Beispiel das gute alte Tremolo, was nichts weiter als eine zyklische Änderung der Lautstärke darstellt. Sehr beliebt ist auch nach wie vor der sog. Slow-Gear-Effekt, der den Ton nach dem Anschlag erst langsam kommen läßt; damit läßt sich die Attack-Zeit beliebig verlängern.

Der Equalizer

Manchmal ist es notwendig, bestimmte Frequenzbereiche eines Sounds anzuheben oder abzuschwächen, um z.B. bestimmte Unregelmäßigkeiten im Frequenzgang eines Bandgeräts oder in der Charakteristik eines Raumes auszugleichen. Hierfür gibt es eine Vorrichtung, mit sich die Lautstärke einzelner Bereiche in einem Klang getrennt regeln läßt, die Equalizer genannt wird. Die Ausführungen reichen hier von der einfachen Zweifach-Klangregelung mit je einem Tiefen- und einem Höhenregler, wie sie häufig in Mischpulten für jeden einzelnen Track vorhanden sind, bis zum 16-Band-Stereoequalizer für ganz exakte Einstellungen des Frequenzgangs. Bei den meisten Modellen läßt sich jedes einzelne Band um jeweils 12 dB anheben oder absenken, was für die meisten Zwecke völlig ausreicht. Nur wenn man extreme Verfremdungseffekte erzielen will, benötigt man einen größeren Regelbereich.

Beim normalen Typ des Equalizers sind die Frequenzbänder festgelegt und können in ihrer Lage nicht verschoben werden; es gibt jedoch auch Modelle, mit denen sowohl die Verstärkung als auch die Position des beeinflußten Frequenzbereichs frei gewählt werden können. Diese Geräte bezeichnet man als parametrische Equalizer. Diese besitzen zwar meist keine so große Anzahl von Bändern, erlauben aber sehr differenzierte Korrekturen und eignen sich daher hervorragend zur Glättung eines welligen Frequenzgangs.

Vibrato

Auch die Beeinflussung der Tonhöhe eines Signals kann recht interessante Effekte hervorbringen. Am bekanntesten ist hier wohl das klassische Vibrato, das eine meist sinusförmige Modulation der Tonhöhe darstellt. Bei gegriffenen akustischen Instrumenten wird hierfür der niedergedrückte Finger einfach hin- und herbewegt, bei Blasinstrumenten erfolgt die Modulation mit Hilfe des Atemdrucks. Um ein Signal nachträglich mit einem Vibrato zu versehen, muß es durch ein Verzögerungsglied geleitet werden, dessen Delay-Zeit zyklisch geändert wird.

Chorus

Der Chorus ist ein dem Nachhall bzw. Echo sehr verwandter Effekt und kann deshalb mit fast jedem Echogerät realisiert werden. Ein Chorus hat im allgemeinen den Sinn, das Signal eines einzelnen Instruments so zu verzögern und zu verändern, daß der Eindruck entsteht, es würden mehrere Instrumente oder Stimmen den gleichen Ton spielen. Erreicht wird das meist mit Hilfe eines Echos von ca. 20 bis 30 ms Verzögerung, wobei das Reflexionssignal noch in seiner Frequenz möglichst frei moduliert wird. Der Chorus-Effekt spielt vor allem im Homerecording und beim Musizieren mit einfachen Mitteln eine große Rolle, da hiermit eine beachtliche räumliche Tiefe erreicht werden kann.

Flanging

Der Flanger ist zumindest technisch gesehen ebenfalls entfernt mit dem Echo verwandt, da es auch auf einer zeitlichen Verzögerung basiert. Die Verzögerung ist hier jedoch so kurz, daß der Effekt nichts mehr mit einem Echo gemein hat. Durch die Mischung des verzögerten mit dem Originalsignal werden vielmehr bestimmte Frequenzanteile hervorgehoben, die den Klang in einer recht eindrucksvollen Weise verfremden können. Die Länge der Verzögerung bestimmt dabei die Lage der verstärkten Frequenzbänder, die sich erwartungsgemäß aus sämtlichen ganzzahligen Vielfachen und Brüchen der Echorate zusammensetzen. Die Verzögerungsdauer wird dann entsprechend moduliert, um den typischen Flanger-Sound zu erzeugen (iieeaaauuuaaaeeiishhh...).

Phasing

Im Phasing findet man ebenfalls eine Methode, gewisse Bänder des zu bearbeitenden Signals hervorzuheben, wobei der Effekt meistens auch zyklisch moduliert wird. Im Gegensatz zum Flanging, das mit einer Zeitverschiebung erreicht wird, beruht der Phasing-Effekt lediglich auf einer Phasendrehung. Genaugenommen entsteht dabei nur ein einziger Frequenzbereich, der dadurch angehoben wird, weshalb Phasing im allgemeinen auch wesentlich »zahmer« klingt als ein entsprechendes Flanging.

Wah-Wah

Das Wah-Wah, das sich vor allem bei Rock-Gitarristen nach wie vor einer großen Beliebtheit erfreut, beruht auch auf der Hervorhebung eines bestimmten Frequenzbandes, nur daß hier ein steiler, variabler Bandpaßfilter verwendet wird. Die Schaltung ist meist in einem Pedal untergebracht, mit dem der Durchlaßbereich des Filters geregelt werden kann. Außer bei Gitarristen ist die Anwendung des Wah-Wahs allerdings eher selten, nur bei Saxophonisten trifft man es noch zuweilen an.

Distorsion

Des Hard-Rock- und Heavy-Metal-Gitarristen liebstes Stück ist jedoch seine Distorsion, auch Fuzz genannt. Diese bewirkt eine gewollte Verzerrung, die durch die Übersteuerung einer Verstärkerschaltung erreicht wird. Dabei tritt ein Spannungs-Clipping auf, das Signal wird also

unten und oben abgeschnitten; eine Rechteck-artige Wellenform entsteht, wodurch ungeradzahlige Oberwellenanteile hervorgehoben werden. Dies hat gleichzeitig eine Dynamik-Kompression zur Folge, da die Lautstärke ja auf ein bestimmtes Level begrenzt, die Verstärkung aber sehr hoch ist. Wenn sich die Eingangslautstärke ändert, so bewirkt das am Ausgang über einen weiten Bereich lediglich eine Änderung des Verzerrungsgrads.

Zumindest die meisten Gitarrenverstärker verfügen bereits über eine Verzerrer-Einrichtung: Oft braucht nur die Vorstufe voll aufgedreht zu werden, oder es existiert eine getrennte Regelung hierfür. Der Anwendungsbereich der Distorsion geht aber noch wesentlich weiter: Im Rock werden gelegentlich auch der Baß und das Keyboard durch einen Verzerrer gejagt; vor allem die in den 60er und 70er Jahren so beliebten Hammond-Orgelsounds verlangen geradezu nach dieser Behandlung, aber auch eine Stimme kann hiermit interessant verfremdet werden (Vorsicht, Rückkopplung!).

Der Korg-A3-Multieffekt-Prozessor

Mit diesem Gerät hat Korg wohl die meisten Wünsche, die man überhaupt an eine Effektbearbeitung stellen kann, in Erfüllung gehen lassen: Der A3 ist ein voll digitaler Effektprozessor, der 19 Grundtypen von Effekten zur Verfügung stellt, von denen jeweils sechs gleichzeitig miteinander verkettet werden können. Für die einzelnen Effekttypen sind teilweise noch mehrere Parameter vorhanden, unter anderem auch das Mischungsverhältnis des bearbeiteten und des Original-(Bypass-)Signals. Hier ein kleiner Überblick der Fähigkeiten des A3:

❑ Hall; hierfür sind zwei verschiedene Bänke vorhanden, nämlich das Early Reflection Program, in dem die für den Raumeindruck maßgeblichen Hallanteile bestimmt werden, und dem eigentlichen Hallprogramm, das für die diffusen Klanganteile zuständig ist.

❑ Echo; dieses bietet eine maximale Verzögerungszeit von 800 ms, wobei die Zeit mit Hilfe einer Feineinstellung sehr gut an den Beat eines Stückes angepaßt werden kann. Es ist beliebig auf die Stereo-Kanäle verteilbar und kann auch frequenzmoduliert werden.

❑ Kompressor und Gate; außer einem Kompressionsalgorithmus besitzt der A3 noch ein sog. Gatter, von der das Signal stummgeschaltet wird, wenn es einen bestimmten (einstellbaren) Pegel unterschreitet.

❑ Panorama und Rotary Speaker; Mit diesen beiden Algorithmen können sehr schöne Bewegungs- und Leslieeffekte erzeugt werden.

❑ Distortion; in dieser Gruppe stehen je zwei Verzerrungs- und Overdrivesimulatoren zur Verfügung.

❑ Pitch Shift; hiermit kann die Tonhöhe eines Signals konstant verschoben werden, maximal bis zu einem Halbton. Vor allem mit Stimmen kann man hier beeindruckende Klänge erzielen.

❑ Exciter; diese Funktion ist eines der Glanzstücke des A3, sie ermöglicht die Erzeugung von beliebigen Obertönen aus dem Eingangsignal. Hiermit lassen sich einzelne Instrumente problemlos in ganze Ensembles verwandeln.

6.2 Die Bandmaschine

Neben dem Sequencer ist wohl das Bandgerät der zentrale Punkt in einem Homerecording-System. Da trotz des Stimmenreichtums der vorhandenen Synthesizer oft nicht alle Tracks einer Komposition gleichzeitig abgespielt werden können, und man außerdem ja auch manchmal akustische Instrumente verwenden will, benötigt man eine Vorrichtung zur getrennten Speicherung möglichst vieler Signale; schließlich will man nach der Erstellung aller Komponenten noch eine Kontrolle über jede einzelne Stimme haben, um ein Mixing vornehmen zu können. Mit anderen Worten, man benötigt eine Mehrspur-Bandmaschine.

Die Ansprüche der Anwender sind hier ebenso unterschiedlich wie die dementsprechenden Preisklassen und Ausführungen der Geräte. Ganz unten auf der Skala befinden sich die preiswerten 4-Spur-Kassettenrecorder, von denen die besseren schon für ca. 1 000 bis 2 000 Mark erhältlich sind. Diese bedienen sich ganz einfach der zwei Stereo-Spurenpaare, die für die beiden Wiedergaberichtungen der Kassette vorhanden sind, weshalb in dieser Weise bespielte Kassetten nur in einer Richtung abgespielt werden können. Die Bandgeschwindigkeit ist hier meistens doppelt so hoch wie bei normalen Kassettengeräten, hauptsächlich um den Frequenzgang und die Gleichlaufeigenschaften zu verbessern. In vielen Fällen auch noch eine einfache Mischvorrichtung integriert, was viel zur Kompaktheit und Übersichtlichkeit eines Homerecording-Systems beiträgt. Die Klangqualität solcher Geräte kann in den meisten Fällen als durchaus befriedigend bezeichnet werden; am meisten ist hier wohl die Wahl des Rauschunterdrückungssystems entscheidend, das ja einerseits ein gutes Signal-Rausch-Verhältnis bewirken, andererseits den Klang so wenig wie möglich beeinflussen und überdies noch preisgünstig sein soll.

Auf der nächsten Stufe findet man die schon deutlich teureren und eher für professionelle Ansprüche konzipierten 8-Spur-Maschinen. Diese besitzen zwar so gut wie nie einen eigenen Mischer, sind aber meist mit sehr ausgefeilten Features im Bereich der Aufnahme- und Wiedergabefunktionen ausgerüstet. Entscheidet man sich für diese Leistungklasse, so muß man sich heute noch darauf einstellen, für Bandgerät und Mixer allein um die 5 000–8 000 Mark hinzulegen. Andererseits ist es auch eine Überlegung wert, daß wir als Amiga-Besitzer ja mit einem Sequencer arbeiten wollen, wobei immer eine Spur für das Synchronisationsignal geopfert werden muß. Bei einem 4-Spur-Gerät bleiben da nur noch kümmerliche drei Tracks übrig, was schon für eine halbwegs komfortable Produktion eines Demo-Bandes einfach nicht mehr ausreicht.

In diesem Abschnitt sollen zunächst die für eine Bandmaschine wichtigen Eigenschaften und Features kurz beleuchtet werden; darauf folgt eine kleine Übersicht über einige der beliebtesten Geräte.

6.2.1 Die Klangeigenschaften einer Bandmaschine

Frequenz- und Phasengang

Ein Tonband ist ein analoges Aufzeichnungsmedium und zeigt infolgedessen leider immer einige typische Mängel bezügliche der Klangqualität und Wiedergabetreue. Eine wesentliche Komponente bildet hier der sog. Frequenzgang; dies ist die Intensität, mit der Signale von einem

bestimmten Eingangspegel auf den verschiedenen Frequenzen wiedergegeben werden. Für ein Aufnahmemedium ist es ideal, wenn die Kennlinie des Wiedergabepegels vollkommen gerade ist, d.h. alle Frequenzen die gleiche Amplitude besitzen. In der Digitaltechnik ist das mühelos zu realisieren, bei einem Tonband jedoch ergeben sich hier fast immer gewisse Unregelmäßigkeiten. Zum einen kann ein Tonband weder beliebig tiefe noch beliebig hohe Frequenzen aufzeichnen, da die Dichte der Magnetpartikel und damit der Polaritätswechsel begrenzt ist; die Wiedergabeintensität sinkt oberhalb des Nutzbereichs steil ab. Zum anderen weisen die meisten Rauschunterdrückungssysteme bestimmte Nichtlinearitäten auf, die dem Frequenzgang ebenfalls eine gewisse »Welligkeit« verpassen.

Alle diese Effekte werden natürlich noch multipliziert, wenn eine Spur auf eine andere kopiert wird. Deshalb ist die Klangqualität auch ausschlaggebend dafür, wie viele »Generationen« von Kopien eines Signals gemacht werden können, ohne daß es auf Grund der Verformung durch die vielen Überspielungen unbrauchbar wird. Guten Geräten liefert im Durchschnitt auch noch die dritte Generation ein annehmbares Ergebnis. Allgemein läßt sich sagen, daß eine Bandmaschine für die hier zugrundegelegten Ansprüche mindestens einen Frequenzgang von 30 bis ca. 16 000 Hz bei einer Abweichung von nicht mehr als ± 3 dB aufweisen sollte.

Eine weitere Eigenschaft vor allem einer Mehrspur-Aufzeichnung ist die Phasenabweichung, die durch den Aufnahme- und Wiedergabeprozeß entsteht. Der Gebrauch von Filtern und anderen Audiokomponenten verursacht z.B. in bestimmten Frequenzbereichen Phasendrehungen, die mit einer Verzögerung gleichgesetzt werden müssen. Auch winzige Abweichungen in der geradlinigen Bewegung des Bandes am Tonkopf können zu kleinen Zeitverschiebungen führen. Für voneinander unabhängige Signale ist das irrelevant, für die zwei Spuren eines Stereosignals aber nicht: Da schon kleinste Zeitverzögerungen (von wenigen Millisekunden) über die räumliche Ortung eines Klanges im Stereo-Panorama entscheidend sind, können sich solche marginalen Effekte bereits sehr störend auswirken. Wenn dasselbe Signal auf mehreren Spuren gleichzeitig vorkommt, ergeben sich durch die Zeitverschiebung zusätzlich unschöne Phasing-Effekte. Auch hier ist wieder sowohl der mechanische Aufbau des Bandlaufwerks sowie die Eigenschaften der vorgeschalteten Elektronik entscheidend.

Der Übersprech-Effekt

Da die Spuren einer Mehrspur-Bandmaschine in der Regel sehr dicht beieinander liegen, kann es vorkommen, daß der Inhalt einer Spur auf einer benachbarten hörbar wird. Dieses Phänomen bezeichnet man als Übersprech-Effekt. Probleme schafft dies allerdings nur, wenn sich dadurch z.B. gelöschte Teile, die in der Endabmischung nicht vorkommen sollen, wieder in den Klang hineinmogeln oder sehr leise Spuren betroffen sind, die später noch einmal kräftig angehoben werden sollen.

Besonders problematisch ist dieser Effekt im Zusammenhang mit den verschiedenartigen Synchronisations-Signalen: Einerseits muß ein Sync-Signal mit einer sehr hohen Pegel aufgesprochen werden, was die Gefahr in sich birgt, daß es auf der Nachbarspur hörbar wird; andererseits kann ein zu lautes Signal auf der Nachbarspur wiederum in das Sync-Signal einstreuen und damit den Synchronizer durcheinanderbringen. Besonders gefährlich sind hier scharfe, percussionartige Klänge mit einem schnellen Anschlag; ein weicher, flächiger String-

Teppich z.B. wird in der Regel keine Störungen bewirken. Eine wichtige Konsequenz aus dieser Sachlage ist, daß eine Sync-Spur grundsätzlich am Rand eines Bandes zu liegen hat, also entweder Spur 1 oder 4 bzw. 8, wo ja immerhin nur eine Nachbarspur existiert. Als zusätzlichen Schutz für den Inhalt der Sync-Spur bieten einige Geräte einen Bandpaßfilter an, der bei der Wiedergabe genau den für das Signal benötigten Frequenzbereich herausfiltern.

6.2.2 Rauschunterdrückungssysteme

Ein Tonband ist, anders als ein digitales Medium, stets von Ungenauigkeiten in der Wiedergabe der aufgezeichneten Signale belastet. So gehorchen lange nicht alle der Metallpartikel in der magnetischen Schicht den Aufforderungen des Aufnahmekopfes, einige richten sich gerade nach Belieben aus. Dies führt bei der Wiedergabe zu einem nicht unerheblichen Rauschen, das bei professionellen Anwendungen inakzeptabel ist und kompensiert werden muß. Auf dem Markt der Mittelklasse-Systeme konkurrieren zur Zeit noch zwei unterschiedliche Verfahren zur Rauschunterdrückung:

DBX NR (Noise Reduction)

Wie störend sich ein bestimmtes Rauschsignal auswirkt, richtet sich wie erwähnt nach dem Verhältnis zwischen Nutz- und Rauschanteil eines Signals. Ein konstantes Rauschen, wie das eines Tonbands, ist demnach in einer lauten Passage so gut wie nicht wahrzunehmen, wohingegen es einem bei einem Pianissimo-Part die ganze Hörfreude verderben kann. Daraus zogen gewitzte Tontechniker bereits in den frühen 70er Jahren eine entscheidende Konsequenz: Das auf das Band aufgenommene Signal sollte möglichst gar nicht mehr so niedrige Lautstärken erreichen, daß das Rauschen überhaupt hörbar wird. Dazu ist es logischerweise notwendig, geringe Lautstärken um einen bestimmten Faktor anzuheben.

Um nun ein in dieser Weise verändertes Signal wieder in den Originalzustand zurückverwandeln zu können, bedarf es hier schon einer gewissen Systematik: Es wird zunächst eine Basislautstärke festgelegt, die man als 0 dB definiert. Hiervon ausgehend werden alle Passagen entsprechend verstärkt, die unterhalb dieses Levels liegen, und überlaute Teile entsprechend gedämpft; dies wird wie bereits angesprochen als Dynamik-Kompression bezeichnet. Bei der Wiedergabe wendet man entsprechend das umgekehrte Verfahren an, d.h., leise Stellen werden wieder in noch leisere und laute Passagen in noch lautere verwandelt (Expansion). Dies bewirkt, daß der unvermeidliche Rauschanteil immer mit der Gesamtlautstärke angehoben und abgesenkt wird, um sein Verhältnis zum Nutzsignal vor allem bei leisen Passagen möglichst günstig zu halten. Da hier alle Frequenzanteile in gleicher Weise berücksichtigt werden, spricht man beim DBX-System von einem sog. Breitband-Kompander.

Dolby B und C

Mit einem Breitband-Kompander läßt sich bereits ein sehr großer Rauschabstandgewinn erreichen, gewöhnlich mehr als 20 dB. Jedoch auch dieses Prinzip ist leider nicht frei von verfahrensimmanenten Mängeln: Wie gesagt wird bei lauten Passagen das Rauschen in der Wiedergabe ebenfalls lauter, was aber in der Regel nicht stören sollte, da der Rauschabstand ja theoretisch hoch genug ist und es eigentlich komplett übertönt werden müßte.

Grau aber ist alle Theorie, die frequenzselektiven Eigenschaften unseres Gehörs können einem hier schnell einen dicken Strich durch die Rechnung machen. Es zeigt sich nämlich, daß gewisse Frequenzen die umgebenden Bereiche besser übertönen können als andere; so werden z.B. tiefe Frequenzen (20–100 Hz) sehr leicht von mittleren (1000–4000 Hz) übertönt, umgekehrt gelingt das jedoch kaum, das dazu nötige Pegelverhältnis wäre hundertfach größer. Beim Breitband-Kompander heißt das: Gibt man eine Passage wieder, die nur aus lauten Bässen besteht, so wird das Bandrauschen mit verstärkt; da aber der rechnerische Störabstand hier nahezu irrelevant ist, wird das Rauschen mit jedem Peak des Baßtons wahrnehmbar, man hört den Kompander »atmen«.

Um diesen lästigen Effekt zu umgehen, ließ man sich in den Dolby Laboratories wiederum etwas einfallen: Statt wie beim Breitband-Kompander die Lautstärke des gesamten Frequenzbereichs als Bezugspegel zu nehmen, teilt Dolby NR diesen in drei (Dolby B) bzw. in vier (Dolby C) voneinander unabhängige Sektoren ein, für die jeweils ein eigener Kompander vorhanden ist. Das verhindert z.B., daß eine Baßtrommel auch den Höhenbereich samt dem darin enthaltenen Rauschen mit anhebt und es dadurch hörbar macht.

Ein weiterer Vorteil der spektralen Aufteilung ist die genaue Anpassung der Reaktionsgeschwindigkeit der einzelnen Kompander an die verarbeiteten Frequenzen. Damit die Behandlung des Signals so wenig wie möglich hörbar wird, muß ein Kompander innerhalb einer sehr kurzen Zeit auf einen Pegelwechsel reagieren und sich auf eine neue Lautstärke einstellen können. Wenn sie dies allerdings zu schnell tun, können tiefe Frequenzen ebenfalls irrtümlich als Pegelsprünge interpretiert werden und den Kompressor in Schwingung versetzen, was natürlich zu unsinnigen Ergebnissen führt. Bei einem Breitband-Kompander muß deshalb sowohl der einbezogene Frequenzbereich nach unten als auch die Reaktionsgeschwindigkeit nach oben stark begrenzt werden. Sind die Frequenzbereiche aber getrennt, so kann die Reaktionszeit für jedes Band einzeln optimal eingestellt werden.

6.2.3 Wichtige Features einer Bandmaschine

Cue- und Auto-Locator

Meistens reichen die normalen Recorder-Funktionen wie Play, Stop, FF und Rewind schon für die grundlegende Produktionsarbeit bei weitem nicht aus. Wenn man z.B. einen schwierigen Instumentalpart einspielen möchte, so werden in aller Regel mehrere Versuche, »Takes« genannt, für ein akzeptables Ergebnis notwendig sein. Dabei ist es natürlich sehr lästig, wenn der Anfang des Parts immer erst wieder von Hand angespult werden muß. Eine kleine Hilfe ist da schon eine Memory-Funktion, die an den meisten preiswerten Geräten vorhanden ist. Diese bricht einen Vor- oder Rückspulvorgang einfach ab, sobald der Bandzähler die Nullposition erreicht hat. Wenn man am Beginn der gewünschten Passage den Zähler auf 0 zurücksetzt, kann man sie nachher bequem wieder hinspulen, ohne dabei ständig auf den Zählerwert achten zu müssen. Dieses Verfahren ist sehr simpel und kann auch mit einem einfachen mechanischen Zählwerk realisiert werden.

Wesentlich mehr bieten hier jedoch Geräte mit einem elektronischen Zählwerk: Bei diesen können meistens eine oder mehrere beliebige Bandpositionen programmiert werden, an denen das Band auf Wunsch gestoppt wird. Aber nicht nur ein Auto-Stopp läßt sich hiermit darstellen, sondern

überdies noch eine Fülle von anderen zeitsparenden Funktionen; z.B. das Auto-Locate, mit dem man eine bestimmte Position anwählen kann, die dann hingespult und wiedergegeben wird; oder auch das kontinuierliche Abspielen eines bestimmten Parts, auf dem man mehrere Takes hintereinander machen will, ohne dabei immer wieder alle Bedienschritte an der Bandmaschine durchführen zu müssen. Durch die große Flexibilität heutiger Mikrocomputer sind hier theoretisch keine Grenzen gesetzt, aufwendig und damit teuer sind hier jedoch die Bedienelemente, weshalb man die meisten dieser Funktionen an den preiswerten Maschinen leider vermißt. Äußerst vielversprechend ist jedoch die Tendenz, daß viele der neueren Geräte, natürlich auch nur in der gehobenen Klasse, mit einer Computer-Schnittstelle ausgestattet sind, über die sämtliche Funktionen des Geräts gesteuert werden können. Sollte dieser Impuls bei den Herstellern von Sequencer-Software auf genügend Resonanz stoßen, sind vor allem im Bereich der kleineren Systeme in nächster Zeit sicher einige überraschende Innovationen zu erwarten.

Punch In

Wenn man mit einigen Stellen in einem aufgenommenen Track nicht zufrieden ist, wird man ihn jedoch meist nicht nochmal ganz einspielen, sondern lediglich stellenweise korrigieren wollen. Dazu ist es aber unbedingt notwendig, daß der musikalische Kontext kurz vor der Passage noch abgehört werden kann, und die Aufnahme auf dem gewünschten Track dann im richtigen Moment einsetzt. Während alle anderen Tracks auf Wiedergabe geschaltet sind, kann man nun die korrigierte Fassung des Parts einspielen. Die Beendigung der Aufnahme (Punch Out) sollte immer auf einer stillen Stelle im Track erfolgen, damit nachfolgende Sounds nicht versehentlich angelöscht werden. Das Ein- und Ausschalten der Aufnahme sollte natürlich keine Clicks oder ähnliches verursachen und darf hinterher auf dem Band nicht zu hören sein.

Um eine solche Funktion zu realisieren, ist bei den Bandgeräten der unteren Preisklassen oft nur ein Fußschalter vorhanden, mit dem die Aufnahme aktiviert werden kann. Viel bequemer ist es natürlich auch hier wieder ein elektronisches Zählwerk, in das die gewünschten Punch-In- und -Out-Positionen einprogrammiert werden können; so kann man sich während der Aufnahme voll auf sein Instrument konzentrieren. Besonders vorteilhaft ist es, wenn statt der Aufnahme der Track einfach nur stummgeschaltet werden kann, so daß für den Punch vorher noch ein paar »Trockenübungen« stattfinden können.

Ping-Pong (Bouncing)

Bei etwas komplexeren Arrangements, die sehr viele Einzelstimmen beinhalten, werden in der Regel auch ebenso viele Bandspuren benötigt, um nachher eine optimale Kontrolle über die Endabmischung zu haben. Da wir aber als arme Hobby-Musiker über keine unerschöpflichen Mengen an Spuren verfügen, wird es oft bald notwendig sein, den Inhalt einiger Tracks zusammenzufassen und auf einen anderen zu überspielen. Dies wird Ping-Pong-Verfahren oder auch Bouncing genannt. Bei den kleinen Bandmaschinen mit integriertem Mischer gibt es dafür meist eine spezielle Schaltweise, in der die Spuren vor dem Überspielen noch durch den Mixer geleitet werden können, damit man die Lautstärken und ggf. den Klang noch einmal nachregeln kann. Externe Mischpulte müssen hierfür ebenfalls einen Modus vorsehen, in dem möglichst sämtliche Features nutzbar sein sollten.

Die Grenzen dieses Verfahrens werden natürlich durch die Wiedergabequalität des Bandes gesetzt. Durch mehrmaliges Überspielen tritt schnell ein Kopiereffekt auf: Der Rauschanteil wird erhöht, Mängel des Rauschunterdrückungssystems werden hörbar, Nichtlinearitäten im Frequenzgang multiplizieren sich. Auch bei einem guten Gerät kann man deshalb sinnvollerweise selten mehr als drei Kopie-Generationen anfertigen, danach werden die Veränderungen meist zu auffällig. In Tests von Mehrspur-Maschinen sind deshalb auch oft die Frequenzgänge verschiedener Generationen angegeben; hieran läßt sich gut ablesen, zu wie vielen Kopiervorgängen ein Gerät fähig ist.

6.3 Synchronisation

Zu guter Letzt müssen noch die beiden zentralen Geräte des Aufnahmesystems, die Bandmaschine und der Sequencer, miteinander synchronisiert werden. Auf Deutsch heißt das, daß sich der eine irgendwie an die Geschwindigkeit des anderen halten muß, um eine zeitliche Verschiebung der beiden zu verhindern. Dies fällt einem Sequencer natürlich leichter, da er nur einen Zeitzähler nachzuführen braucht und ohnehin durch ein externes Clock-Signal gesteuert werden kann; bei einer Bandmaschine dagegen muß hierfür der Antriebsmotor nachgeregelt werden, was ziemlich aufwendig ist.

Die einfachste Lösung dieses Problems besteht daher in einem Gerät, welches ein Synchronisationssignal auf eine Spur der Bandmaschine aufzeichnet, das mit der Position der Wiedergabe im Song korreliert. Dies wird als erstes in einem Vordurchlauf auf das Band geschrieben, von wo es bei jedem weiteren Aufnahmevorgang abgelesen und interpretiert wird. Bei allen weiteren Vorgängen wird der Synchronizer zum Master des Sytems, d.h., sämtliche Sequencer werden von seinem Clock-Signal gesteuert. Dazu liest er den Sync-Ton eine Weile lang ab und ermittelt daraus die Stelle im Song, an der sich das Band gerade befindet. Hierauf wird er meist einen Song-Position-Pointer-Befehl senden, um die internen Zeiger des Sequencers korrekt einzustellen; durch einen Continue-Befehl wird dieser dann am richtigen Punkt eingestartet.

Auch hier gibt es jedoch verschiedene Verfahren, die sich in ihrer Flexibilität und natürlich auch im Preis deutlich unterscheiden. Dies sind die drei am meisten verwendeten Prinzipien:

Klick-Track

Die eindeutig billigste Methode für die Synchronisation mit einem Band ist die sog. Klick-Spur, bei der lediglich im Abstand von Sechzehntel- oder Zweiunddreißigstel-Noten laute Einzelimpulse auf den Track aufgesprochen werden. Der Hardwareaufwand hierfür ist denkbar gering, weshalb man dieses Verfahren auch heute noch z.B. bei preiswerten Drum-Machines antrifft. Die Arbeit damit ist aufgrund seiner geringen Flexibilität jedoch ziemlich umständlich: Da es sich hier nur um einfache Klicks handelt, die nichts über die Song-Position verraten, müssen Band und Sequencer für jede Aufnahme von Anfang an neu gestartet werden, was ein vernünftiges Arbeiten nahezu unmöglich macht. Aus diesem Grund soll hier nicht weiter darauf eingegangen werden.

FSK-Sync

Um einen Sequencer korrekt einstarten zu können, ist es also notwendig, daß das Sync-Signal ständig Auskunft über die gerade gespielte Stelle im Song gibt. Dazu müssen die Informationen

im Signal so verschlüsselt werden, daß sie durch die Bandaufzeichnung nicht verstümmelt werden; der Ausgang einer seriellen Schnittstelle ist dafür nicht geeignet, da in den Daten hin und wieder auch längere Perioden mit gleichem Pegel auftreten und ein analoges Tonband keine Gleichspannungen verarbeiten kann. Deshalb bedient man sich hier oft des sog. FSK-Verfahrens (FSK = Frequency Shift Keying), bei dem die beiden binären Level, 0 und 1, vor der Aufnahme in zwei bestimmte Frequenzen umgewandelt werden. Diese Methode wird auch in Akustikkopplern für die Datenübertragung über Telefon eingesetzt. Leider ist die maximale Datenrate auf ca. 300 Baud begrenzt, weshalb die Beat-Auflösung hier nicht besonders genau sein kann.

Das Verfahren hat aber noch einige andere gewichtige Nachteile: Da z.B. die Frequenzumtaster, die das Signal wieder interpretieren sollen, genau auf eine bestimmte Frequenz eingestellt sind, darf die Abspielgeschwindigkeit des Bands auf keinen Falll verändert werden. Das macht Tonhöhen-Korrekturen für die Abstimmung z.B. auf einen bestimmten Flügel unmöglich. Aus dem gleichen Grund können die meisten Geräte keine FSK-Spuren lesen, die von anderen Geräten aufgenommen wurden. Auch kann eine solche Sync-Spur nicht geschnitten oder anderweitig korrigiert, sondern muß vor der Aufnahme aller anderen Tracks erstellt werden und ist dann nicht mehr zu verändern.

SMPTE- und EBU-Timecode

Das Bedürfnis nach der Synchronisation von Geräten ist jedoch noch wesentlich älter als MIDI-Schnittstelle und Sequencer; in der Filmproduktion müssen ja immmer Bild und Ton aufeinander synchronisiert und immer wieder geschnitten werden. Schon 1972 definierte die US-amerikanische »Society of Motion Picture and Television Engineers« (SMPTE) ein Format für ein Sync-Signal, das sich an der Anzahl der Bilder eines Films oder Videogeräts orientiert. Dies sind in der US-Norm 30, in der eropäischen Norm aber nur 25 Bilder pro Sekunde; da man aber einen einheitlichen Standard etablieren wollte, übernahm die European Broadcasting Union (EBU) das Aufzeichnungs-Format der SMPTE, nur auf die hier übliche Bildanzahl modifiziert.

Da der Timecode auf so unterschiedlichen Medien wie Bandmaschine und z.B. Videorecorder aufgebracht werden muß, existieren hierfür zwei Methoden: die Unterbringung der Code-Daten in der Austastlücke des Video-Bildes (VITC = Vertical Interval Timecode), und die Aufzeichnung als Audio-Signal auf einem Band (LTC = Longitudinal Timecode). Erstere wird im allgemeinen nur von sehr teueren Synchronizern unterstützt, die wohl nur im professionellen Bereich zu finden sind. Im Bereich der Bandsynchronisation, wo sich SMPTE ebenfalls bereits zum Standard entwickelt hat, sind jedoch bereits sehr preiswerte Geräte erhältlich.

Das Datenformat des SMPTE-Codes ist aufgeteilt in die Stunden, Minuten, Sekunden und Einzelbilder (Frames), die seit der Filmklappe bzw. dem Beginn des Songs verstrichen sind. In jedem einzelnen Frame müssen deshalb sämtliche Daten der Position enthalten sein. Ein Frame wird dazu wiederum in 80 Zeiteinheiten aufgeteilt, in denen je ein Datenbit untergebracht werden kann. Physikalisch funktioniert das so: 80mal pro Frame wird grundsätzlich ein Polaritätswechsel, also eine steile positive oder negative Flanke auf das Band gebracht, um einen Grundtakt zu geben. Erfolgen genau zwischen diesen Wechseln weitere Flanken, so bedeutet das ein 1-Bit, kein Wechsel bedeutet ein 0-Bit. Der zeitliche Abstand der Bits beträgt im EBU-Format 0.5 ms, was demenstprechend auch die Genauigkeitsgrenze eines solchen Systems darstellt. Für den

Timecode selbst werden jedoch nicht alle 80 Bit eines Frames benötigt: 32 davon sind sog. User-Bits und können vom Anwender beliebig gesetzt werden.

Nun wird man sich berechtigterweise fragen, wie denn bitte die starre Einteilung in Sekunden und Frames mit mit der ja stets variablen Geschwindigkeit eines Sequenzers vereinbar ist. Das muß wiederum vom Synchronizer erledigt werden, der in diesem Fall von Anfang an die Rolle des Masters, also des Taktgebers via MIDI-Clock übernehmen muß. Das Tempo kann dann nicht mehr am Sequencer, sondern nur noch am Synchronizer eingestellt werden. Wenn also z.B. das Band an irgendeiner Stelle gestartet wird, empfängt der Synchronizer den Timecode, errechnet aus dem eingestellten Tempo die Stelle im Song und gibt eine entsprechende Song-Pointer-Botschaft mit anschließendem Continue-Befehl aus. Das bedeutet leider auch, daß während der Arbeit an einem Song nicht einfach der Synchronizer gewechselt werden kann, da sich die realen Tempowerte bei den einzelnen Geräten geringfügig unterscheiden.

Für viele Zwecke reicht diese Technik aus; kommen im Song jedoch Tempowechsel vor, so müssen diese entweder mühsam von Hand programmiert werden, oder der Synchronizer muß mit der Fähigkeit ausgestattet sein, sich diese selbständig zu merken, einem sog. Learn-Mode. Diesen besitzen jedoch bei weitem nicht alle Modelle, unter anderem da dies eine ganze Menge Computerleistung erfordert. Optimal ist es, wenn der Synchronizer bereits ein Teil des Sequencer-Paketes ist und damit sämtliche Berechnungsvorgänge im Computer selbst durchgeführt werden können.

6.4 Die Aufnahme

Der Sync-Track

Am Beginn einer jeden Aufnahme steht immer das Aufbringen einer Sync-Spur auf das Band. Zuerst stellt sich die Frage nach der Nummer der Spur, die verwendet werden soll; wie bereits erwähnt, sollte dies zur Minimierung des Übersprech-Effekts unbedingt eine Randspur sein. Viele Bandmaschinen sehen hierfür bereits eine Spur vor, auf der z.B. die Rauschunterdrückung abgeschaltet werden kann, um das Signal so wenig wie möglich zu verfremden. Das nächste Problem ist die Einstellung des Aufnahmepegels, mit der das Signal aufgesprochen wird. Wird dieser zu hoch gewählt, so wird der Sync-Ton auf der Nachbarspur hörbar; nimmt man den Sync dagegen zu leise auf, kommt es unter Umständen zu Lesefehlern. Bei professionellen Bandmaschinen mit sehr geringem Übersprechen wäre ein Pegel um –10 dB ein guter Mittelwert, bei preiswerteren Geräten etwas darüber: ca. –3 dB.

Dann kann man an die Aufnahme des Sync-Signals herangehen. Bei Klick-Spur und FSK-Sync wird dazu das Band auf Record gestellt und dann der Sequenzer gestartet. Dabei sollte man jedoch möglichst alle Spuren des Sequencers stumm schalten, damit die Clock-Befehle nicht durch andere Busaktivitäten verzögert werden. Ein SMPTE-Synchronizer erledigt diesen Vorgang selbstständig, der Sequencer ist in diesem Fall nicht beteiligt.

Bei der weiteren Aufnahmearbeit treten allerdings Probleme auf, wenn die Spur direkt neben der Sync-Spur bespielt werden soll, während von dieser synchronisiert wird. Um ein Tonband

magnetisieren zu können, muß der Aufnahmekopf ein sehr starkes Magnetfeld erzeugen. Aus diesem Grunde tritt im Moment der Aufnahme ein wesentlich stärkeres Übersprechen auf, als das bei der Wiedergabe der Fall ist. Der Effekt kann so ausgeprägt sein, daß eine Synchronisation unter diesen Umständen unmöglich wird. Die Spur kann in diesem Fall nur noch dann für Aufnahmen genutzt werden, wenn der Sync nicht gleichzeitig gelesen werden muß.

Aussteuerung

Als nächster Schritt müssen vor der ersten Aufnahmephase alle Spuren auf die Lautstärke der beteiligten Instrumente eingepegelt werden. Grundsätzlich gilt natürlich auch hier, daß die höchste Aussteuerung auch den besten Störabstand bewirkt. Aus diesem Grund sollte man den Pegel wirklich soweit anheben, wie es das verwendete Bandmaterial erlaubt. Da die meisten modernen Aufnahmeköpfe recht gutmütig auf Übersteuerung reagieren, kann man hier ziemlich hart an die Grenze gehen. Bei den recht preiswerten Chromdioxid-Bändern sollte z.B. mühelos ein Aufnahmepegel von +3 dB zu verkraften sein, bei Metallbändern noch weit darüber.

Overdubbing

Nun kann die eigentliche Produktionsarbeit beginnen. Abhängig von den vorhandenen Keyboards und Expandern, oder besser deren Stimmenanzahl, wird man die Stimmen des Songs in Gruppen aufteilen, die zusammen eingespielt werden sollen. Dabei ist es vorteilhaft, wenn diese gleich so auf den Spuren zu liegen kommen, wie sie für die Endabmischung gebraucht werden, um möglichst keine Überspielungen vornehmen zu müssen und so ein gutes Stück Klangqualität zu retten. Deshalb sollte das vorhandene Stimmenpotential immer voll genutzt werden, damit der Song in möglichst wenigen Durchgängen und einem geringen Verbrauch an Bandspuren eingespielt werden kann.

In vielen Fällen wird es jedoch nicht möglich sein, sämtliche MIDI-Stimmen in einem Take in den Kasten zu bekommen; dann muß die Aufnahme auf mehrere Durchgänge verteilt werden. Möchte man darüber hinaus noch akustische Instrumente oder Vocals hinzufügen, so ist dies ohnehin nur nacheinander möglich. Das weitere Vorgehen hängt jetzt sehr stark von der Anzahl der zur Verfügung stehenden Bandspuren ab: Besitzt man nur eine 4-Spur-Maschine, so muß man sich die Reihenfolge der einzelnen Aufnahmephasen gut überlegen, da die Anzahl der verbleibenden Spuren ja mit jedem abgeschlossenen Vorgang geringer wird. Möchte man hier z.B. seine eigene Stimme übereinanderlegen, so muß man diese auf getrennte Spuren aufnehmen und dann im Ping-Pong-Verfahren auf eine andere Spur übertragen; das geht aber nur, wenn mindestens drei Spuren frei sind, zwei für die Aufnahme und eine weitere als Zielspur beim Überspielen. Gleichzeitig möchte man aber auch beim Einspielen solcher Instrumente bereits eine Begleitung haben; zumindest ein Taktbezug muß hergestellt sein. Ein Ausweg ist hier vielleicht, wenn man alle Vorgänge ganz am Anfang durchführt, die durch die Anzahl der Durchgänge sehr viele Spuren beanspruchen, und den Sequencer dabei nur zur Begleitung mitlaufen läßt.

Die Endabmischung

Die letzte Phase in der Produktion eines Songs ist die Abmischung des Mehrspur-Bandes auf die üblichen zwei Stereo-Spuren, hier wohl am besten auf ein Kassettendeck. Das Mischpult muß hierfür in der Lage sein, jede Spur auf die beiden Stereokanäle beliebig zu verteilen und dem Sound damit eine Ortung im Stereobild zu geben. Dies besorgt in der Regel der sog. Panning-Regler. Leider ist das nur mit den einzelnen Spuren möglich; sind auf einer Spur mehrere Instrumente, so können diese natürlich nicht mehr getrennt geregelt werden. Sowohl die Positionen im Stereobild als auch die Pegel der einzelnen Instrumente sind damit nicht mehr nachkorrigierbar. Wenn man also mehrere Tracks im Ping-Pong-Verfahren auf einen anderen überspielt, so muß man schon hier einen genauen Mix vornehmen, was aber in der Regel nur mit einiger Übung und Erfahrung zu befriedigenden Ergebnissen führen wird.

Gelegentlich möchte man aber doch möglichst allen aufgenommenen Instrumenten eine individuelle Stereo-Ortung geben, und dennoch mit Hilfe des Ping-Pong-Verfahrens Spuren einsparen. Außerdem besitzen die meisten Synthesizer ohnehin schon Stereo-Ausgänge, über die eingebaute Effekte (wie z.B. Chorus) wesentlich besser klingen als in Mono. Hierfür könnte man folgendes Verfahren vorschlagen, obwohl auch dieses mindestens eine 8-Spur-Maschine voraussetzt: Zuerst bestimme man zwei Spuren, auf die der Stereo-Pre-Mix aufgebracht werden soll. Auf die restlichen Spuren zeichnet man dann die einzelnen Stimmgruppen auf, die einzeln im Stereobild positioniert werden sollen. Anschließend überspielt man alle bis dahin aufgenommenen Spuren auf die beiden gewählten Stereo-Tracks, wobei man mit der Aussteuerung einer jeden Quelle auf den beiden Spuren das Panning regelt. Bei der Endabmischung wird dann jede der so entstandenen Stereo-Spuren auf einen der beiden Ausgangskanäle gelegt.

Abschließend muß noch einmal daran erinnert werden, daß wir ja mit einem Sequencer-System arbeiten, und die Bandmaschine hier im Prinzip nur eine periphere Rolle spielen sollte. Für eine optimale Ausnutzung der vorhandenen Geräte sollte man natürlich immer nur diejenigen Stimmen auf Band bringen, bei denen das unbedingt nötig ist, wie akustische Instrumente oder Vokalstimmen; gerade das macht ja die Flexibilität der Kombination Sequencer/Bandmaschine aus. Im MIDI-orientierten Studio wird man deshalb in der Regel alle Stimmen der vorhandenen Synthies und Expander ausreizen, bevor man einen Overdub vornimmt. Für die Endabmischung ist es deswegen überaus nützlich, wenn der Mixer zusätzlich zu den Bandspuren noch ein paar Tracks mehr besitzt; z.B. kann ein 8-Spur-Mixer für die Arbeit mit einer 4-Spur-Maschine durchaus sinnvoll sein. Dann aber ist es ohne weiteres möglich, alle verfügbaren Synthi-Stimmen während der Poduktionsarbeit immer in Realtime mitlaufen zu lassen und erst beim Endmix mit einzubringen, was natürlich ein großes Plus an Klangqualität bringt.

Das SMUSPlayer Developers Kit

Teil

B

Einführung – die Entstehung des IFF-Formats 1

1.1 Anforderungen an ein neues Format

Die erstaunlichen Leistungen der Home- und Personalcomputer neuerer Generation hat die Entwicklung einer großen Anzahl von Anwenderprogrammen in den Bereichen Textverarbeitung, Grafik und Musik etc. verursacht. Da alle diese Programme, die mit ihnen bearbeiteten Daten in irgendeiner Form abspeichern und wieder laden können müssen, bedienen sie sich hierfür mehr oder weniger standardisierter File-Formate, deren Vielfalt in ähnlichem, hier aber weniger erfreulichen Umfang zugenommen hat. Selbstverständlich ist es so gut wie unmöglich, für alle erdenklichen Kombinationen vorhandener Programme Umwandlungstools mitzuliefern, so daß dem User in aller Regel die gewünschte Verwendung »fremder« Datensätze verwehrt bleibt. (Dies gilt speziell für Programme, deren Fileformat lediglich ein Memory-Dump der Daten im Arbeitsspeicher darstellt.)

Die Forderung nach einem Datenformat, welches den Ansprüchen heutiger und vor allem zukünftiger Programmideen- und Entwicklungen Rechnung zu tragen vermag, und gleichzeitig Verwendbarkeit der Datensätze in artverwandten Programmen gewährleistet, scheint die einzige Konsequenz in dieser Situation zu sein. Dies stellt jedoch hohe Anforderungen an die Flexibilität des Formatkonzepts:

Erstens ist es praktisch nicht möglich, sämtliche Erfordernisse und Eigenschaften zukünftiger Entwicklungen vorauszusehen und ihnen gerecht zu werden. Es müssen daher stets Möglichkeiten zur Erweiterung, d.h. zur Hinzufügung weiterer Elemente und Details vorhanden sein. Um hierfür nicht auf eine »zentrale formatdefinierende Autorität« angewiesen zu sein, dürfen diese Erweiterungsmöglichkeiten jedoch keinen Einfluß auf die restliche Struktur des Formats ausüben, was nur durch vollkommene Modularität erreicht werden kann. Das Ziel sollte hierbei sein, jedem Programm zu ermöglichen, fremde Datensätze einzulesen, elementare Datengruppen zu interpretieren, nicht identifizierte Teile zu überlesen, fehlende Komponenten durch Voreinstellungen (Default-Werte) zu ersetzen und so die Daten unter Verwendung des restlichen Kontextes in die jeweilige Repräsentation im Arbeitsspeicher umzuwandeln.

Zweitens bedeutet die Realisierung eines Programmes, welches sich nach einem möglichst allgemein gehaltenen Standardformat richten soll, oft einen nicht unerheblichen Mehraufwand gegenüber dem oben erwähnten MemoryDump-Verfahren. Um diesen in erträglichen Grenzen zu halten, benötigen Softwareentwickler möglichst standarisierte Zugriffsprozeduren und Unterprogramme, welche genausoleicht modifizierbar und erweiterbar sein sollten wie das Datenformat selbst. Die hierdurch gewonnene Modularität jedoch ermöglicht eine weitgehende Isolation der Anwendung von den Eigenschaften des verwendeten Formates und erlaubt dem Programmierer eine hiervon unabhängige Konzeption seiner Idee.

Von diesen Überlegungen angeregt, entschloß sich 1985 die Firma Electronic Arts, den »EA IFF 85«-Standard zu entwickeln und allen Amiga-Programmierern zur Vereinheitlichung ihrer Produkte anzubieten. Vor allem durch auf dem Apple MacIntosh existierende Formate inspiriert, orientiert sich dieser Standard hauptsächlich an der Architektur des MC68000-Prozessors, welcher auch das Herz des Amiga bildet.

1.2 Datentypen und Speicherbedarf

1.2.1 Adressenanpassung (Alignment)

Eine der wesentlichen Eigenschaften des MC68000-Prozessors ist es, daß alle Datentypen, die größer sind als 1 Byte (Wörter, Langwörter) nur an geraden Adressen liegen dürfen. Eine Nichtbeachtung dieser Konvention veranlaßt den 68000 zum Sprung in eine Exception und damit den Amiga zum Absturz. Aus diesem Grund müssen sämtliche Daten, die Wort- oder Langwortbreite aufweisen, an Wortgrenzen liegen und Byte-Daten, wie z.B. Text, mit einem Ergänzungsbyte (Pad-Byte) abgeschlossen werden, sofern sie auf einer ungeraden Adresse enden. Das Pad-Byte sollte beim Schreibvorgang stets mit einem 0-Wert belegt werden, es wird jedoch empfohlen, dies beim Lesevorgang nicht als unbedingt gegeben vorauszusetzen.

1.2.2 Numerische Werte

Im IFF-Standard sind, obwohl der 68000 Byte-, Wort- wie Langwortdaten sowohl mit als auch ohne Vorzeichen verarbeiten kann, nur die folgenden Typen zugelassen:

UBYTE	8 Bit	ohne Vorzeichen
UWORD	16 Bit	ohne Vorzeichen
WORD	16 Bit	vorzeichenbehaftet
LONG	32 Bit	vorzeichenbehaftet

Diese werden z.B. in C (Aztec,Lattice-C) wie folgt definiert:

```
typedef unsigned char UBYTE;
typedef unsigned short UWORD;
typedef short WORD;
typedef long LONG;
```

1.2.3 Textdaten (Characters)

Sämtliche zugelassenen Character-Werte in Zeichenketten und Identifiern (siehe unten) befinden sich auf der ASCII-Tabelle zwischen den Werten " " (Space, hex $20) und "~" (Tilde, hex $7F). Eine Ausnahme bilden Linefeed (neue Zeile, hex $0A) und Carriage Return (Wagenrücklauf, hex $0D). Die restlichen Werte von hex $0 bis hex $1F haben in IFF keine definierte Bedeutung. Der Bereich von hex $7F ist für zukünftige Erweiterungen reserviert.

1.2.4 Identifier

Zur Kennzeichnung bestimmter Datengruppen dient im IFF-Standard ein vom Apple MacIntosh übernommenes Prinzip, welches sich 4 Byte (= 32 Bit) breiter sog. »Type-ID« bedient und so der Arbeitsweise des 68000 besonders entgegenkommt. Sie bestehen aus einer Aneinanderreihung von 4 ASCII-Zeichenwerten aus dem Bereich " " (Space, hex $20) und "~" (Tilde, hex $7F). Die C-Definition lautet:

```
typedef char ID[4];
```

Type-ID sollten nicht von einem Leerzeichen (Space, hex $20) angeführt werden, an allen sonstigen Positionen sind Leerzeichen erlaubt. Da es nötig sein kann, Type-ID in DOS-Dateinamen zu verwenden (z.B. als Extension), dürfen sie keine DOS-Interpunktionszeichen (wie : und /) sowie keine Kleinbuchstaben enthalten. Letzteres ist nötig, da der Vergleich zwischen Type-ID stets case-sensitiv ist (d.h. Groß- und Kleinbuchstaben werden unterschieden), das DOS Groß- und Kleinschreibung jedoch ignoriert, was sonst zu Problemen führen kann. Man beachte, daß Type ID sowohl in der oben beschriebenen Weise als auch als *LONG* (32 Bit vorzeichenbehaftet) definiert werden können, da sie nur Byte-Werte bis höchstens $7F enthalten. Das oberste Bit ist also nie gesetzt, was die Berücksichtigung des Vorzeichens unnötig macht. Um eine Type-ID in eine *LONG*-Konstante umzuwandeln, verwendet man in C üblicherweise das folgende Makro:

```
#define MakeID(a,b,c,d) ((a)<<24 | (b)<<16 | (c)<<8 | (d))
```

1.3 Das Chunk-Prinzip

Ein File-Format muß sich in erster Linie nach den Eigenschaften der File-Verwaltung richten, von der es verwendet wird, in unserem Fall des Amiga-DOS. Um jedoch weitgehende Unabhängigkeit vom System und damit eine gewisse Portabilität zu gewährleisten, beschränkte man sich beim IFF-Standard auf die beiden grundlegenden Formen des Dateizugriffs: sequentieller und wahlfreier Zugriff. Beim Ersterem werden die Daten in ihrer »natürlichen« Reihenfolge nacheinander geschrieben oder gelesen (im Amiga durch die DOS-Funktionen *Write()* und *Read()* realisiert). Viele Operationen, wie z.B. »Piping«, d.h. die Weitergabe von Datensätzen unter mehreren Programmen, erfordern diese Technik. Bei Letzterem wird zuerst die Position der gewünschten Daten innerhalb der Datei angewählt und erst dann der Schreib- bzw. Lesezugriff ausgeführt (Amiga-DOS-Funktion *Seek()*), was gegenüber dem erstgenannten Verfahren meist einen hohen Geschwindigkeitsvorteil bietet.

Um diesen wirklich zu nutzen, scheint die Verwendung von festen Positionen innerhalb der Datei angeraten, was sich jedoch bei ersterem Verfahren von vornherein verbietet. Um sequentiellen Zugriff zu ermöglichen, ohne der Methode des wahlfreien Zugriffs allzuviel von ihrer Effektivität zu nehmen, wurde hier ein Mittelweg beschritten.

1.3.1 Identifier und Längenzähler

Das dem IFF-Format zugrundeliegende Ordnungsprinzip ist die Organisation jeglicher in einem File enthaltenen Daten zu sogenannten »Chunks« (engl. : Klotz, Stück). Ein Chunk besteht aus

einem Identifier (in Form einer der oben erwähnten Type-ID), einem Längenzähler vom Typ *LONG* (32-Bit-vorzeichenbehaftet) und den jeweiligen Chunk-spezifischen Daten (siehe Abb. 1.1). Die C-Definition lautet. (Sie dient lediglich zur Veranschaulichung, wegen der fehlenden Größenangabe würde sie kein Compiler akzeptieren:)

```
typedef struct {
    ID ckID;
    LONG ckSize; /* sizeof(ckData) */
    UBYTE ckData[/* ckSize */];
} Chunk;
```

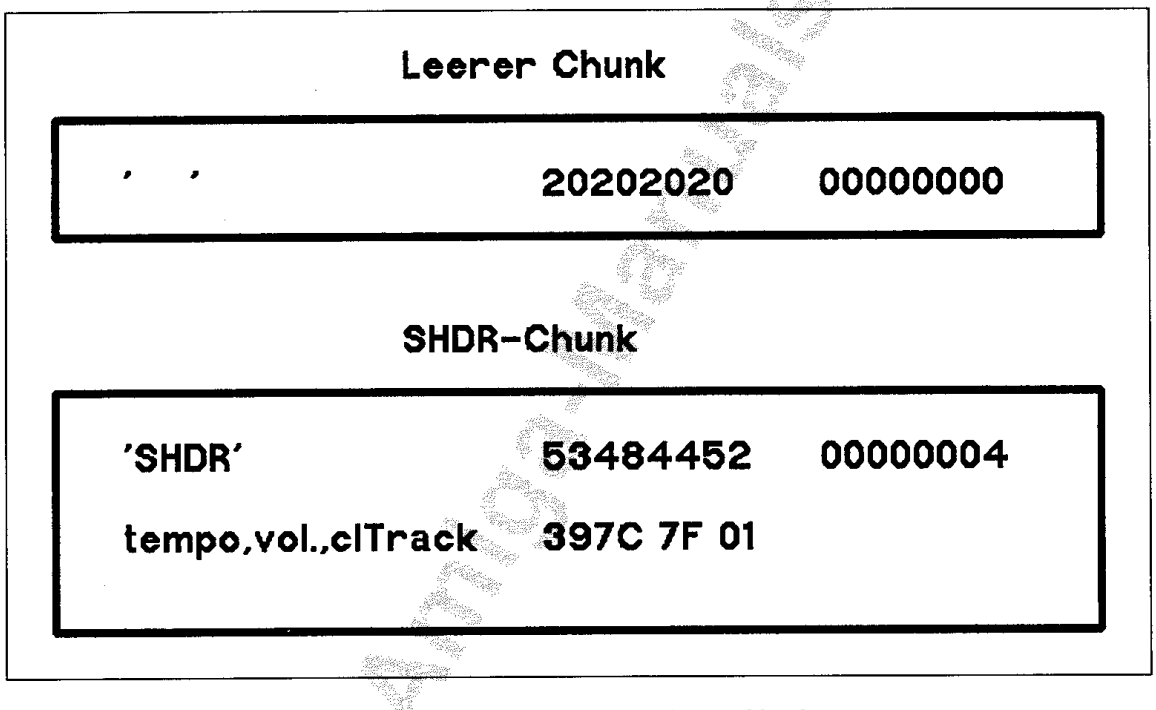


Abb. 1.1: SSHdr-Chunk und leerer Chunk

Der Identifier *ckID* bestimmt den Typ des Chunks und damit das Format sowie den Verwendungszweck der Chunk-Daten. Eine Leseprozedur muß also sämtliche Type-ID, welche das Programm verwenden soll, mit *ckID* vergleichen und, wenn die Suche erfolgreich war, den Chunk interpretieren, andernfalls aber überspringen. Die folgenden Type-ID sind in IFF für den Gebrauch als »Strukurelemente« reserviert:

"FORM"	"FOR1"..."FOR9"	Form-Chunk
"LIST"	"LIS1"..."LIS9"	List-Chunk
"CAT "	"CAT1"..."CAT9"	CAT-Chunk
"PROP"		Property-Chunk

Diese 23 ID bilden die syntaktische Struktur des IFF-Formats, wobei »FOR1« bis »FOR9« nur als zukünftige Versionsangaben vorgesehen sind (analog »LIS1« bis »LIS9« und »CAT1« bis

Texte von Amiga-Manuals-Website

»CAT9«). Innerhalb dieser Chunks dürfen weitere FORM-, LIST-, und CAT-Statements verschachtelt sein, wobei FORM stets eine fest definierte Anordnung von Daten- und Property-Chunks (oder weiterer FORMs, LISTs und CATs) darstellt, CAT und LIST jedoch nur eine lose Aneinanderreihung der anderen Strukturelemente sind. Da das SMUS-IFF-Format selbst jedoch ausschließlich aus »FORM«-Statements besteht (siehe unten), soll hier auf die anderen Elemente nicht weiter eingegangen werden.

Der Wert von *ckSize* gibt die Länge des Datenfeldes *ckData* in Bytes an. Wenn dieser ungerade ist, folgt auf den Chunk ein 0-Byte, um den nächsten Chunk wieder an einer Wortgrenze beginnen zu lassen. Da es in *ckSize* nicht mitgezählt wird, ergibt sich die Gesamtlänge eines Chunks aus dem auf 2 aufgerundeten Wert von *ckSize* plus 8 Bytes für die Einträge *ckID* und *ckSize* selbst. In einem »leeren« Chunk ist also die Gesamtlänge gleich acht und *ckSize* gleich null.

Diese Anordnung ermöglicht die folgenden Zugriffstechniken:

Beim Lesevorgang kann einerseits der Chunk-Kopf (*ckID* und *ckSize*) eingelesen und *ckID* mit gesuchten Type-ID-Strings verglichen werden. So dann wird *ckSize* aufgerundet und die dadurch bestimmte Anzahl von Bytes eingelesen, unabhängig davon ob der Chunk interpretiert werden konnte oder nicht (sequentieller Zugriff). Andererseits kann im Falle eines »fremden« Chunks die gegenwärtige Position im File auf den nächsten Chunk gerichtet werden, indem man (per wahlfreiem Zugriff) »ckSize« zu dieser addiert, um dort mit der Interpretation fortzufahren.

Beim Schreibvorgang hat man nun die Möglichkeit, entweder zuerst sämtliche erforderlichen Werte in einem vorherigen Durchlauf zu bestimmen, um dann den ganzen Chunk auf einmal zu übergeben, oder aber zuerst die eigentlichen Daten zu schreiben und dann wieder zur Position von *ckSize* zurückzukehren, um diese zu berichtigen.

Der Aufbau eines Chunks wird im folgenden durch diese Standardschreibweise ausgedrückt:

```
Chunk ::=   ID # { UBYTE* }   [0]
```

»Chunk« ist der hierbei zu definierende Chunk-Typ, »ID« die Type-ID, die ihm zugeordnet werden soll, und »#« repräsentiert *ckSize*. Hierauf schließt sich das (von Typ zu Typ unterschiedliche) Datenfeld an, welches im Bedarfsfall durch [0], d.h. ein Pad-Byte, abgeschlossen wird.

1.3.2 Der FORM-Chunk

Wie oben erwähnt, gibt es im IFF-Standard eine Klasse von Chunks, die ein IFF-File als solches identifizieren und seinen Aufbau strukturieren. Sie stellen gewissermaßen syntaktische Befehle dar, ähnlich denen für Strukturdefinitionen in einigen Programmiersprachen (wie z.B. *struct* in C).Der hierbei am häufigsten verwendete ist der FORM-Chunk, der folgendermaßen definiert wird:

```
FORM        ::= "FORM" #{ FormType (LocalChunk | FORM | LIST | CAT)* }
FormType    ::= ID
LocalChunk  ::= Chunk | Property
```

Da wir uns wie gesagt auf die Erfordernisse des SMUS-IFF-Formats beschränken wollen, welches die Gruppen LIST und CAT nicht verwendet, können wir die Definition vereinfachen:

```
FORM        ::= "FORM" #{ FormType (LocalChunk | FORM)* }
FormType    ::= ID
LocalChunk  ::= Chunk | Property
```

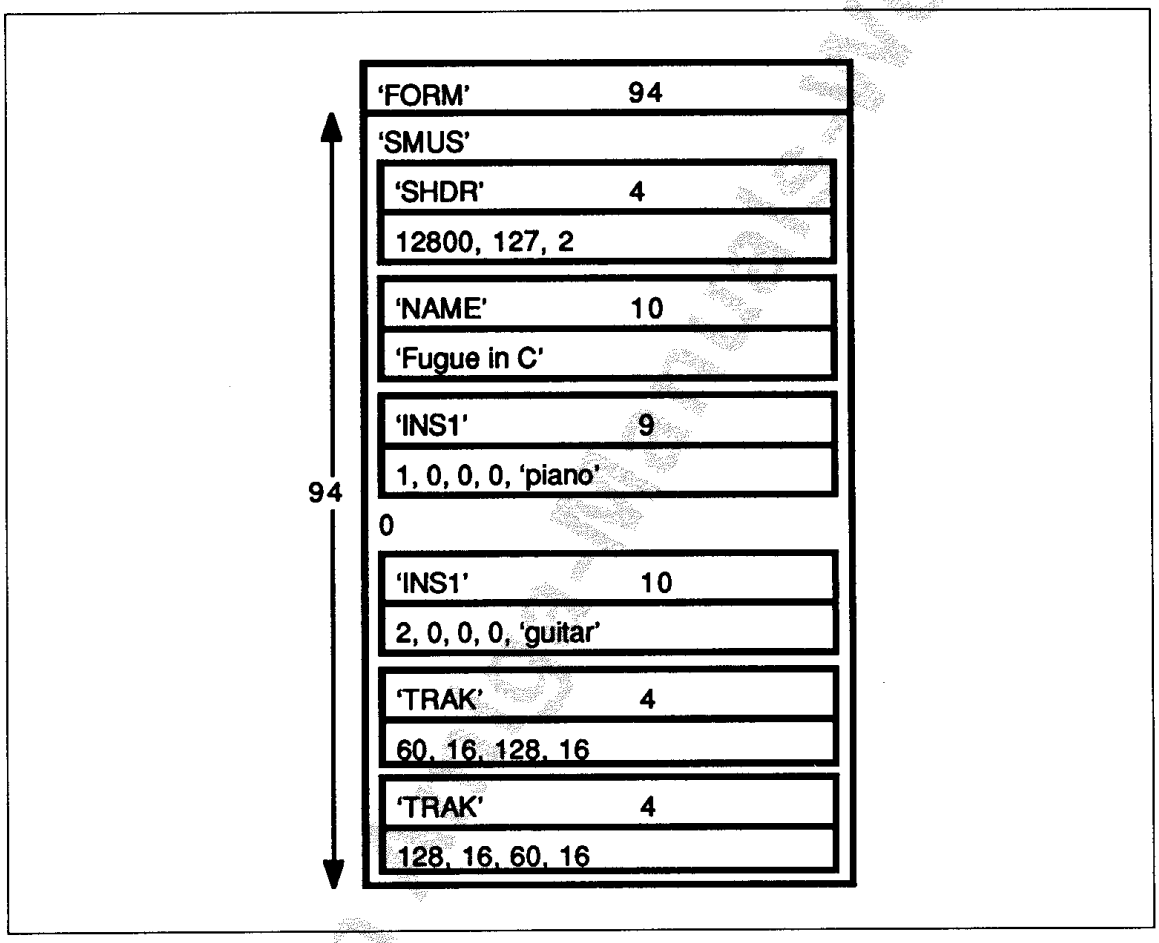


Abb. 1.2: Form-Chunk

Hieraus ist ersichtlich, daß jedes IFF-File von einem FORM-Statement umschlossen sein sollte, damit sich die Gesamtlänge des Files aus dem *ckSize*-Wert errechnen läßt und so alle vorhandenen Daten erkannt werden können. Dieser Wert ist für die Gesamtlänge der Anordnung verbindlich. Tritt aufgrund eines Fehlers beim Schreibvorgang eine Differenz zwischen ihm und der Summe der einzelnen Chunk-Längen auf, so muß er stets als mögliches Ende der Datei betrachtet werden, da es sonst zu Fehlermeldungen der Dateiverwaltung kommen kann.

Das erste Element des FORM-Chunk-Datenfeldes ist wiederum eine Type-ID, welche den Typ des FORM-Chunks und damit den Typ der in ihm zu erwartenden weiteren Chunks bestimmt. Hier eine kleine Auswahl solcher FORM-Typen:

SMUS	simple musical score	(einfaches Musikformat)
8SVX	8-Bit sampled voice	(Musikinstrument-Format)
ILBM	interleaved bitmap	(Grafikformat)
FTXT	formatted text	(Textformat)
FNTR	raster font	(Zeichensatzformat)

...

Anschließend beginnen die FORM-Typ-spezifischen Chunks, deren Gesamtheit eine dem Typ eigene Syntax bilden. Leseprozeduren sollten allerdings unbedingt so flexibel sein, die einzelnen Komponenten in beliebiger Reihenfolge lesen zu können, um gegen Abweichungen unempfindlich zu sein. Zu bemerken ist, daß jeder dieser sog. »LocalChunks« nur im Zusammenhang mit dem jeweiligen FORM-Typ definiert ist, d.h. ID können innerhalb verschiedener FORM-Anordnungen unterschiedliche Bedeutungen haben. So kommt man in der Namensgebung für neue Komponenten einer FORM nicht so schnell in Verlegenheit.

Die in einer FORM auftretenden Chunks lassen sich grob in zwei Kategorien einteilen: Daten-Chunks, die z.B. Bitmap-, Noten-, oder Sounddaten enthalten, und »Eigenschafts«-Chunks (Properties, siehe oben). Letztere haben die Aufgabe, die in den Daten-Chunks enthaltenen Informationen näher zu spezifizieren, wie z.B. die Farben einer Grafik anzugeben, Tempo und Lautstärke eines Liedes zu bestimmen und ähnliches mehr. Da es nötig sein kann, eine bestimmte Eigenschaft innerhalb einer FORM mehrmals zu verändern, muß man einen Geltungsbereich für die jeweiligen Property-Chunks definieren. Die diesbezügliche Regel lautet: Alle Angaben gelten so lange, bis sie explizit geändert werden. So könnte man z.B. für die einzelnen Stimmen eines Musikstückes verschiedene Lautstärken bestimmen, indem man zuerst die Lautstärke A angibt, dann die Daten für Stimme A folgen läßt, dann Lautstärke B angibt und so fort. Treten in einem File jedoch mehrere FORM-Statements auf, so sollte man bei der Interpretation eines neu begonnen FORM-Chunks wieder auf die jeweiligen Default-Einstellungen zurückgreifen.

Für weitere Informationen über den IFF-Standard und die ihm untergeordneten Formate siehe [Addison-Wesley, The Amiga ROM Kernel Manual: Exec, App. B-29 ff / Addison-Wesley: The Amiga R.K.M. Includes&Autodocs]

1.3.3 LIST, CAT und Properties

Außer von einem FORM-Statement kann ein IFF-File auch von einem LIST- oder CAT-Statement umschlossen werden. Diese dienen der Aneinanderreihung von weiteren FORM-, LIST- oder CAT-Chunks, wobei FORM einen klar definierten Satz von Chunks beinhaltet, LIST und CAT jedoch letztlich nur Aneinanderreihungen von FORM-Chunks sind.

CAT-Chunks

Ein CAT-Chunk besteht aus der ID »CAT« und dem ChkSize-Feld, welches die Länge der gesamten Anordnung angibt. Hierauf schließt sich, ähnlich wie beim FORM-Chunk, eine weitere ID an, die den Inhalt des CAT-Chunks charakterisiert:

```
CAT          ::= "CAT " #{ ContentsType (FORM | LIST | CAT)* }
ContentsType ::= ID (abstrakter Datentyp oder ein Hinweis auf einen)
```

Die ID »ContentsType« sollte im Normalfall identisch sein mit der der in der CAT-Anordnung befindlichen FORM-Chunks, oder gleich 0000 sein, wenn mehrere verschiedene FORM-Typen enthalten sind. CAT-Chunks sollten nicht ineinander verschachtelt sein, sondern eher hintereinander angeordnet werden.

LIST-Chunks

LIST-Chunks unterscheiden sich von CAT-Chunks nur in der Hinsicht, als daß eine Möglichkeit gegeben ist, globale (also für alle enthaltenen Chunks gültige) Properties zu definieren. Dies geschieht mit Hilfe von PROP-Chunks (siehe unten). Ein LIST-Chunk ist so definiert:

```
LIST          ::= "LIST" #{ ContentsType PROP* (FORM | LIST | CAT)* }
ContentsType  ::= ID
```

PROP-Chunks

PROP-Chunks enthalten Property-Chunks, die für die FORM-Statements in einer LIST-Anordnung gelten sollen. So könnte man z.B. verschiedenen SMUS-Scores in einem LIST-Chunk ein für alle verfügbares Instrumentenset zuordnen, in dem man dieses in einem PROP-Chunk unterbringt. Die Definition eines PROP-Chunks lautet:

```
PROP        ::= "PROP" #{ FormType Property* }
FormType    ::= ID
```

FormType zeigt hierbei an, für welche Art von FORM die aufgeführten Property-Chunks bestimmt sind. Alle in einem PROP-Chunk aufgeführten Properties haben nur Gültigkeit innerhalb des sie enthaltenden LIST_Chunks. Außerdem kann eine Propety innerhalb eines FORM-Chunks geändert werden. Zum Beispiel kann in einer Liste von SMUS-FORMs, deren Instrumente durch einen PROP-Chunk definiert wurden, das Instrumentenset eines einzelnen Scores dadurch umdefiniert werden, indem in diesem andere Instrumenteinstellungen als Property-Chunks enthalten sind. Außerhalb dieses FORM-Chunks gelten wieder die im PROP-Chunk definierten Instrumente.

Taken from Amiga-Manuals-Website

SMUS IFF – Simple-Musical-Score-Interchange-File-Format

2

Der SMUS-IFF-Standard ist ein typischer Vertreter der IFF-FORM-Familie. Vollkommen modular aufgebaut, eignet er sich sowohl zum Abspeichern reiner Musikdaten eines Kompositionsprogrames, als auch zur Integration in andere IFF-Files, so z.B. einer Video-Animation mit musikalischer Untermalung. Er wurde von Electronic Arts ursprünglich für Spieleprogrammierer entwickelt, die Musik und Soundeffekte in ihre Projekte integrieren, hierbei aber möglichst auf existierendes Material oder zumindest Fragmente davon zurückgreifen wollen. Verwendet wird er inzwischen in leistungsfähigen Musik-Editorprogrammen wie Deluxe Music Construction Set oder Aegis Sonix, die dem Anwender eine große Vielfalt an Ausdrucks- und Variationsmöglichkeiten bieten.

Beim Entwurf des SMUS-Formats orientierte man sich an den Bedürfnissen »klassischer« Musikstücke, welche im allgemeinen durch recht wenige Elemente, nämlich Notendauern (ganze, halbe, Viertelnoten etc.) und dazugehörige Instrumente sowie Lautstärke und Tempo dargestellt werden können. Dies erlaubt zwar keine absolut freie Kompositionstechnik, wie in vielen Formen moderner Musik oft erwünscht, genügt aber für die meisten Anwendungen auf der Ebene heutiger Unterhaltungs- und natürlich auch Computermusik.

2.1 Definitionen

2.1.1 Events

Da Musik als eine Abfolge akustischer Ereignisse anzusehen ist, liegt es nahe, als Grundeinheit für musikalische Daten das »Ereignis« oder Event zu definieren. Um jedoch nicht für jedes dieser Ereignisse immer alle eventuell benötigten Parameter (wie Tempo, Instrument, Lautstärke eines Tones) mit angeben zu müssen, ist es zweckmäßig, ein Klasse von Events zu etablieren, mit denen sich alle erforderlichen Werte einstellen lassen. Diese Einstellungen müssen, analog der bei den Properties erwähnten Regel, bis zur nächsten ausdrücklichen Änderung Gültigkeit haben. Ihr Kontext sollte so gewährleisten, daß jedes auftretende Event durch die zwei Parameter »Ereignis-Art« und »Dauer« vollständig beschrieben wird.

Im SMUS-Format führte man die folgenden Typen von Events ein:

SNote	Note-Event
SID_FirstNote	erste Note
SID_LastNote	letzte Note
SID_Res	Pause
SID_Instrument	neues Instrument
SID_TimeSig	Taktmaß setzen

SID_KeySig	Tonart setzten
SID_Dynamic	Lautstärke setzen
SID_MIDI_Chnl	Midi-Kanal angeben
SID_MIDI_Preset	MIDI-Presetwert angeben
SID_Clef	Notenschlüssel setzen (Oct. 88)
SID_Tempo	Tempo setzen (Oct. 88)

2.1.2 Tracks

Bei den meisten elektronischen wie auch bei vielen akustischen Musikinstrumenten ist die Tonerzeugung an »Stimmen« (Tracks) gebunden, d.h. physikalische Einheiten, die jeweils nur zur Erzeugung eines Tones fähig sind. Bei einer Gitarre z.B. ermöglichen die sechs Saiten maximal sechs Töne gleichzeitig zu spielen, ähnlich bei den meisten Synthesizern mit ihren Oszillatoren oder bei den vier Digital/Analog-Wandlern des Amiga. Um eine Anordnung solcher Einheiten mit den benötigten Informationen zu versorgen, bieten sich grundsätzlich zwei Wege an:

❑ Die Speicherung sämtlicher Daten in einem einzigen Informationsstrom, unter Angabe der Stimme, der ein Event zugeordnet werden soll. Dies erlaubt z.B. ein sequentielles Abspielen von Daten, die nicht vollständig im Arbeitsspeicher Platz finden und nur bei Bedarf von Zeit zu Zeit nachgeladen werden. Der Speicherbedarf ist hier jedoch relativ hoch, da für die Trackzuweisung bei den im SMUS-Format maximal definierten 255 Tracks ein ganzes Byte pro Event benötigt wird.

❑ Die Aufteilung der Daten in separate Informationsströme, von denen jeweils einer einem bestimmten Track zugeordnet ist. Die Numerierung erfolgt gemäß der physikalischen Reihenfolge der Track-Daten in der Datei, wobei man darauf achten sollte, daß die wichtigen Stimmen zuerst auftreten, und erst dann die Begleitungsstimmen folgen. Dadurch werden in einem Musikstück (im folgenden Score genannt), welches mehr Tracks enthält, als die abspielende Software bzw. Hardware bearbeiten kann, lediglich die Stimmen mit geringerer Priorität ignoriert, um den Charakter des Stückes möglichst wenig zu verfälschen.

Hauptsächlich aus Gründen der Speicherplatzersparnis und der einfacheren programmtechnischen Realisierbarkeit, entschied man sich beim SMUS-IFF-Standard für die letztgenannte Methode.

2.1.3 Instrumente

Der wichtigste Parameter für ein akustisches Ereignis ist nach Tonhöhe und Tondauer wohl die Klangcharakteristik, d.h. das Verhältnis zwischen den Teilschwingungen eines Tones sowie der Lautstärkeverlauf. Diese hängen natürlich von der jeweiligen Wiedergabeeinheit ab, deren Eigenschaften in den meisten Fällen auch noch in einem weiten Bereich variiert werden können. Es bedarf also schon einer sehr komplexen Anordnung von Parametern, um einen Klang umfassend beschreiben zu können. Man unterscheidet im IFF-Format zwei Fälle:

❑ Die Klangwiedergabeeinheit befindet sich innerhalb des Computers, auf dem das Abspielprogramm (Player-Programm) läuft, kann also direkt angesteuert werden (wie hier die D/A-Wandler des Amiga). In diesem Fall werden sämtliche Werte, mit denen die interne Hardware

versorgt werden muß, wiederum in einem bestimmten Format zusammengefaßt und mit Hilfe der computereigenen Fileverwaltung archiviert (siehe Kapitel »Instrumentenformate«).

❑ Der Ton soll von einem externen, über ein MIDI-Interface angeschlossenen Synthesizer erzeugt werden. Dann muß der Player lediglich die innerhalb des externen Gerätes gewünschte Stimme (MIDI-Kanalnummer) und Klangeinstellung (MIDI-Presetwert) an die Schnittstelle übergeben. Wenn das geschehen ist, können beliebige Folgen von Notenwerten, bestehend aus Tonhöhe und Dauer, an den Synthesizer ausgegeben werden (siehe Kapitel »Was ist MIDI«).

Wie die Einstellungen von Lautstärke und Tempo, so wird auch das gegenwärtig von einem Track benutzte Instrument durch ein dafür definiertes Event angegeben. Dieses Event enthält nun die Nummer des gewünschten Instruments als Referenz auf eine von sog. »INS1«-Chunks gebildeten Liste innerhalb der SMUS-FORM. Die Nummer ist hierbei die Ordnungsnummer in der physikalischen Reihenfolge der INS1-Chunks. Jeder Track eines Scores muß entweder auf eines dieser Instrumente gesetzt sein oder, falls im Track nach dem Start noch keine Instrumentenangabe erfolgt ist oder das gesuchte Instrument nicht gefunden wurde, mit einer Voreinstellung (Default-Instrument) belegt werden.

2.2 Standard-SMUS-Chunks

In diesem Kapitel sollen nun die Chunk-Typen, die die Syntax der SMUS-FORM bilden, im einzelnen erläutert werden. In Abb. 2.1 sehen Sie den Aufbau eines SMUS-Files, wie es von einem beliebigen Score-Editor erzeugt werden könnte. Das File umschließende FORM-Statement wird gefolgt von dem (notwendigen) Property-Chunk »SHDR«, den (optionalen) Instrument-referenzen »INS1« sowie den eigentlichen Notendaten in Form der »TRAK«-Chunks. In diesem Beispiel handelt es sich um eine C-Dur-Tonleiter, welche durch das Instrument »Piano« gespielt werden soll.

2.2.1 Der SHDR-Chunk

Dieser Chunk erscheint in den meisten Fällen als erster in einer SMUS-FORM. Er enthält eine *SScoreHeader* genannte Struktur, die in C folgendermaßen definiert wird:

```
define ID_SMUS MakeID('S','M','U','S')
define ID_SHDR MakeID('S','H','D','R')

typedef struct {
   UWORD tempo;      /* Tempo in (Viertelnoten/Minute) * 128 */
   UBYTE volume;     /* Gesamtlautstärke von 0-127 */
   UBYTE ctTrack;    /* Anzahl der "TRAK"-Chunks */
} SScoreHeader;
```

Das Strukturelement *tempo* gibt die Abspielgeschwindigkeit sämtlicher Tracks im Score an. Abhängig davon, ob ein Event zur Tempoänderung während der Wiedergabe definiert ist (siehe unten), stellt dies entweder einen verbindichen Wert für die gesamte Dauer der Wiedergabe dar, oder nur einen Anfangswert, der bei Bedarf geändert wird. Die Angabe bezieht sich auf die Anzahl

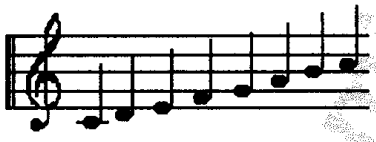
'FORM'	464F524D	0000005A
'SMUS'	534D5553	
'SHDR'	53484452	00000004
tempo,vol.,ctTrack	397C 7f 01	
'NAME'	4E414D45	0000000F
'C-Dur-Tonleiter'	432D4475722D546F 6E6C656974657200	
'INS1'	494E5331	00000009
reg.,type,data1,2	00 00 00 00	
'Piano'	5069616E6F00	
'TRAK'	5452414B	00000018
SID_TimeSig: 4/4	82 1A	
SID_KeySig: C-Dur	83 00	
SID_Dynamic: 0 dB	84 7F	
SID_Instrument: #0	81 00	
	3C 02 3E 02 40 02 41 02 43 02 45 02 47 02 48 02	

Abb. 2.1: C-Dur-Tonleiter

der Viertelnoten pro Minute, multipliziert mit 128. Ein Wert von 1 bedeutet also 1 Viertelnote in 128 Minuten, der höchste mit einer *UWORD*-Variable darstellbare Wert, 65536, ergibt eine Geschwindigkeit von 512 Viertelnoten pro Minute. Die meisten Playerprogramme werden diesen Bereich jedoch nicht ganz ausnutzen, und ihn in der Regel auf einen sinnvollen Abschnitt von ca. 30–300 Viertelnoten pro Minute beschränken.

Das Element *volume* bezeichnet die Lautstärke, mit der alle Tracks wiedergegeben werden. Mit diesem Wert müssen die individuell eingestellten Lautstärken der jeweiligen Tracks skaliert werden, um die effektive Lautstärke eines Tones zu erhalten. Der Wertebereich umfaßt 0 bis einschließlich 127, wobei 0 unendlich leise (–[unendlich] dB) und 127 Vollausschlag (0 dB) bedeutet.

Das Element *ctTrack* schließlich enthält die Anzahl der Tracks eines Scores, sprich die Anzahl der in der SMUS-FORM enthaltenen »TRAK«-Chunks (siehe unten).

2.2.2 Die optionalen Text-Chunks NAME, (c), AUTH, ANNO

Im SMUS-Format gibt es eine Klasse von Chunks, die zusätzliche Informationen über den Score wie Autor, Erstellungsdatum oder evtl. Urheberrechtsschutz enthalten können. Die Datenfelder dieser Chunks bestehen aus den im IFF-Standard üblichen Character-Werten aus dem Bereich " " (Space, hex $20) bis "~" (Tilde, hex $7F). Die Größe *ckSize* gibt hier die Anzahl der Bytes wieder, die von dem jeweiligen Text-String belegt wird.

Der NAME-Chunk enthält den Namen des entsprechenden Scores. Man beachte, daß sich dieser Name keineswegs mit dem Dateinamen des jeweiligen Scores decken muß, wobei der im NAME-Chunk angegebene Name den »eigentlichen« Namen des Scores darstellt. Am besten ist es also, beim Entwurf eines Score-Editors darauf zu achten, daß sich diese beiden Angaben niemals widersprechen können.

Der (c)-Chunk kann einen Copyright-Vermerk enthalten, wobei z.B. ein (c)-Chunk, gefolgt von den Character-Werten »1990 by Nicolaus Wirsing« ganz einfach die Standardformulierung "(c) 1990 by Nicolaus Wirsing" bedeutet.

Da es sich bei den beiden letztgenannten Chunks um Property-Chunks handelt, ist eine evtl. Mehrfachangabe redundant, d.h. nur der jeweils zuletzt aufgetretene Chunk wird berücksichtigt. Sie sollten außerdem höchstens eine Länge von 256 Byte aufweisen, da dies auf den meisten Maschinentypen die maximale Länge für Character-Strings (wie z.B. Dateinamen) darstellt.

Der ANNO-Chunk schließlich enthält einen beliebigen Kommentar. Dieser kann eine beliebige Länge aufweisen, in den meisten Programmen dürfte jedoch aus praktischen Gründen der Wertebereich einer *WORD*-Variablen (d.h. $2^{15} = 32768$) die Obergrenze sein.

Die ID dieser Chunks werden in C durch folgende Makros eingeführt:

```
#define ID_Name MakeID('N','A','M','E')
/* enthält einen char[]-Vektor (den Namen des Scores) */

#define ID_CopyRight MakeID('(','c',')',' ')
/* enthält einen char[]-Vektor (Copyright-Hinweis) */

#define ID_AUTH MakeID('A','U','T','H')
/* enthält einen char[]-Vektor (den Namen des Autors etc.) */

#define ID_ANNO MakeID('A','N','N','O')
/* enthält einen char[]-Vektor (Erstellungsdatum o.Ä.) */
```

2.2.3 Der INS1-Chunk

Im INS1-Chunk werden die Instrumente definiert, die in einem Score verwendet werden sollen. Einfache Playerprogramme können diese Angaben ignorieren und statt dessen eigene eingebaute Instrumente verwenden, höherentwickelte werden sie jedoch berücksichtigen, um die musikalische Charakteristik eines Scores möglichst wenig zu verfälschen. Wie erwähnt gibt es im SMUS-Format zwei Möglichkeiten, ein Instrument zu spezifizieren: Erstens durch Angabe eines Namens, nach dem entweder im Score-File selbst gesucht wird, in Form einer eigenen Instrumenten-FORM, oder auch in einem speziellen Instrumentenverzeichnis (üblicherweise ein Dateiverzeichnis), und zweitens durch die MIDI-Parameter eines angeschlossenen Sythesizers. Diese beiden Möglichkeiten werden in der *RefInstrument*-Struktur, die den Inhalt eines INS1-Chunks bildet, berücksichtigt. Die C-Definition lautet:

```
#define INS1_Name   0
#define INS1_MIDI   1

typedef struct {
   UBYTE register;      /* Instr.- Registernummer */
   UBYTE type;          /* Typ: Name oder MIDI */
   UBYTE data1,data2;   /* abhängig von type */
   CHAR  name[];        /* Instr. -Name */
} RefInstrument;
```

Das Element *register* enthält die Registernummer des durch diesen Chunk definierten Instruments. Wenn in einem Track das Event *SID_Instrument* abgearbeitet wird, so wird die darin enthaltene Registernummer mit den Registernummern aller vorhandenen INS1-Chunks verglichen, bis ein solcher mit der gleichen Nummer gefunden wird (siehe Tracks). Dadurch wird das gewünschte Instrument eindeutig bestimmt, ohne sämtliche dafür notwendigen Daten in einem File mehrmals auftreten lassen zu müssen.

Das *type*-Element bestimmt, ob ein Instrument durch einen Namen definiert wird oder ob es zu der Klasse der über MIDI wiederzugebenden Instrumente gehört. Steht hier der Wert *INS1_Name*, so bedeutet das, daß das Instrument in der oben beschriebenen Weise über den im Feld *name* angegebenen Namen gesucht wird. In diesem Fall sollten die Elemente *data1* und *data2* mit dem Wert 0 belegt werden. Ist hier aber der Wert *INS1_MIDI* enthalten, so wird *data1* als die MIDI-Kanalnummer und *data2* als MIDI-Presetwert angesehen, und von nun an ein externes Gerät mit den Daten des Tracks versorgt.

Das *name*-Feld sollte unabhängig vom Inhalt des *type*-Elements mit einem gültigen Namen versehen sein (oder zumindest einem Leerstring, bestehend aus einem 0-Byte), da weniger hochentwickelte Playerprogramme die MIDI-Option wahrscheinlich ignorieren und in jedem Fall nach einem Instrument des angegebenen Namens suchen werden.

2.2.4 Der TRAK-Chunk

Der TRAK-Chunk nun enthält die eigentlich abzuspielenden Noten in Form von 2 Byte breiter sog. *SEvent*-Einträge, die in ihrer zeitlichen Reihenfolge abgespeichert vorliegen. Die TRAK-

Chunks repräsentieren die verschiedenen Stimmen des wiederzugebenden Stückes, für jede Stimme ein TRAK-Chunk. Es ist auch eine Möglichkeit vorgesehen, mehrere Töne gleichzeitig, also einen Akkord innerhalb eines einzelnen TRAK-Chunks anzugeben, was theoretisch auch eine Beschreibung eines Scores in einem einzigen TRAK-Chunk ermöglichen würde. Diese Option wird jedoch aus den im Kapitel 2.1.2 genannten Gründen nur von sehr wenigen Playerprogrammen genutzt.

Die in diesen Chunks enthaltenen *SEvent*-Einträge bestehen aus jeweils einer 8 Bit breiten Typ-Angabe sowie einem 8 Bit breiten Datenfeld, welches typspezifische Daten enthält. Die C-Definitionen lauten:

```
#define ID_TRAK MakeID('T','R','A','K')

/* Definition von SEvent */
typedef struct {
   UBYTE sID;
   UBYTE data;
} SEvent;

/* mögliche sID-Typen */
#define SID_FirstNote      0
#define SID_LastNote       127

/* sID`s im Bereich von SID_FirstNote bis
SID_LastNote stellen Notenwerte dar. */

#define SID_Rest           128
#define SID_Instrument     129
#define SID_TimeSig        130
#define SID_KeySig         131
#define SID_Dynamic        132
#define SID_MIDI_Chnl      133
#define SID_MIDI_Preset    134
#define SID_Clef           135
#define SID_Tempo          136

/* Die sID-Werte von 144 bis 159 sind für
Instant-Music SEvents reserviert (!!!),
die restlichen Werte bis 254 für zukünftige
Erweiterungen des Standards. */

#define SID_Mark     255
/* Definiert als Ende-Zeichen für
Repräsentationen der Daten im Arbeitsspeicher */
```

Im den folgenden Kapiteln sollen Eigenschaften und Verwendungszweck dieser *SEvent*-Typen genauer erläutert werden.

Anmerkung: Alle weiteren möglichen *SEvent*-Typen sind optional. Score-Editorprogramme brauchen sie nicht zu verwenden, und Playerprogramme können sie ohne Gefahr ignorieren.

Tech-Tools Amiga-Manuals-Archive

Die Events »Note« und »Pause«

Die in einem TRAK-Chunk auftretenden *SEvent*-Typen lassen sich grob in drei Klassen einteilen:

a) Noten-Events
b) Pausen
c) technische Events

Die Events mit den *sID*-Werten von 1 bis einschließlich 126 gehören zur Klasse a), d.h., sie bedeuten für das Playerprogramm einen direkt wiederzugebenden Ton mit entsprechender Tonhöhe und Dauer. In diesem Fall ist die Einbettung in die *SEvent*-Struktur folgendermaßen definiert:

```
typedef struct {
   UBYTE           tone; /* MIDI-Tonnummer von 0 bis 127 */
   unsigned chord  :1,   /* Akkord? */
   tieOut          :1,   /* 1 bedeutet gebundene Note */
   nTuplet         :2,   /* 0 = normale Note, 1 = Triole,
                         /* 2 = Quintole, 3 = Septole */
   dot      :1,          /* punktierte Note? */
   division :3;          /* 0 = ganze Note, 1 = halbe Note,
                            2 = Viertelnote, 7 = 128tel Note */
} SNote;
```

Anmerkung zur Implementation: Einige C-Compiler unterstützen keine sog. Bitfelder, wie sie in der letzten Definition verwendet werden. Lattice-C z.B. stellt Bitfelder grundsätzlich in einer Breite von 32 Bit dar, was bei der *SNote*-Struktur zu einer Länge von insgesamt 5 Bytes führen würde. Sollten Sie einen solchen Compiler verwenden, so müssen Sie anstatt der vorhergehenden Schreibweise die Definition des *SEvent* selbst verwenden, unter Zuhilfenahme der unten angeführten Konstantendefinitionen, welche die entsprechenden Bitmasken und Verschiebungswerte enthalten.

Das Strukturelement *tone* enthält die sog. MIDI-Tonnummer des wiederzugebenden Tones. Dieser Wert kann aus dem im MIDI-Standard definierten Bereich von 0 bis 127 stammen, wobei 0 die tiefste, 127 die höchste mögliche Tonhöhe bedeutet. Die Numerierung der Tonhöhen orientiert sich an der direkten Reihenfolge der Halbtöne im hörbaren Bereich. Deswegen kann durch Division der MIDI-Tonnummer durch 12 (da eine Oktave 12 Halbtöne enthält) die Oktave und durch den ganzzahligen Rest dieser Division die relative Tonhöhe innerhalb der Oktave ermittelt werden. Beispielsweise bedeutet hier ein Wert von 60 ein eingestrichenes C.

Der sog. Notenwert (die Länge des Tones) wird durch die Kombination der Strukturelemente *nTuplet*, *dot* und *division* bestimmt. *division* gibt hierbei die »Grundlänge« einer Note an: ganze, halbe, Viertel-, Achtelnote bis hinunter bis zur Hundertachtundzwanzigstelnote.

Das Element *dot* bestimmt, ob es sich hierbei um eine sog. »punktierte« Note handelt. Diese wird im Verhältnis zu einer gleichwertigen nicht punktierten Note genau eineinhalbmal länger gespielt.

nTuplet gibt an, ob eine sog. n-Tole der Ordnung 3, 5 oder 7 vorliegt. Eine Triole besitzt $^2/_3$ der Länge einer normalen Note, eine Quintole $^4/_5$, eine Septole $^6/_7$ des normalen Notenwerts. Ge-

bräuchlich innerhalb der konventionellen Kompositionstechniken (vor allem in der Unterhaltungsmusik) sind jedoch nur die Triolen, wie z.B. im Jazz oder Blues, wo sie die Voraussetzung für die diesen Stilrichtungen eigene »Phrasierung« bilden. Quintolen- und Septolenwerte werden deshalb von den meisten Playerprogrammen ignoriert.

Die effektive Länge eines Tones ergibt sich also nach der Formel:

Grundlänge	*	*Punktierung*	*	*n-Tole*
(1,1/2,1/4,..)		(1,3/2)		(1,2/3,4/5,6/7)

Für Playerprogramme empfiehlt es sich jedoch, diese drei Strukturelemente zu einem Wert zusammenzufügen (durch entsprechende Bit-Maskierung des *SEvent.data*-Feldes) und über eine Tabelle den effektiven Wert für einen Zeitzähler o.ä. zu gewinnen.

Das *chord*-Bit bestimmt, ob eine Note mit der nachfolgenden Note im Track einen Akkord bildet, d.h. zur gleichen Zeit gespielt wird. Das bedeutet, daß die Zeitverzögerung in der Abfolge der Töne des Tracks für dieses Event auf 0 gesetzt wird, weswegen der Notenwert dieses Events ignoriert wird. Die Notenwerte der Events eines solchen Akkords sollten jedoch aus Sicherheitsgründen stets dieselben sein, um dem Akkord eine eindeutig definierte Länge zu geben. Auch sollte der »wichtigste« Ton des Akkords (wahlweise der »Grundton«, der tiefste, oder der »Melodieton«, der höchste) an letzter Stelle stehen, um simpleren Playerprogrammen, welche keine chorded Notes unterstützen, das Wiedergeben des für die Klangcharakteristik am meisten signifikanten Tones zu erlauben. Das erste *SNote*-Event, in dem das *chord*-Bit gleich 0 ist, beendet die Folge von Akkordtönen und gibt meist die eigentliche Spieldauer des Akkords an. Überflüssig zu sagen, daß innerhalb einer Folge von zu einem Akkord zusammengefaßten *SEvent*s keine anderen Events, wie Instrumentänderungen etc. vorkommen dürfen, da sich sonst verschiedene Parameter eines Tracks auf ein und denselben Zeitpunkt beziehen würden, was höchstwahrscheinlich zu Problemen in der Steuersoftware führen wird.

Das *tieOut*-Bit zeigt an, ob die Note mit der Nächsten gebunden werden, d.h. eine Note mit der Länge der Summe der beiden einzelnen Noten gespielt werden soll. Es können beliebig viele Noten auf diese Weise zu einem einzigen Ton zusammengebunden werden, vorausgesetzt, sie haben alle die gleiche Tonhöhe, d.h., daß die *tone*-Elemente der einzelenen *SEvent*s die gleichen Werte aufweisen. Sollte dies bei einem *SEvent*, dem ein gebundenes vorausging, nicht der Fall sein, so muß die Bindung ignoriert werden und ein neuer »Tastenanschlag« (Attack genannt) für die neue Tonhöhe durchgeführt werden.

Auch Akkorde lassen sich mit dem *tieOut*-Bit um beliebige Werte verlängern, wobei dieses in jedem einzelnen *SEvent* der Akkord-Kette auf 1 gesetzt sein muß. Hierbei ist es praktikabel, wenn die Reihenfolge der Akkord-Noten jeweils dieselbe ist, so daß ein Playerprogramm in dieser Reihenfolge nach den Entsprechungen der Noten in beiden Akkorden suchen kann, ohne jede Note des einen Akkords mit jeder des Anderen vergleichen zu müssen. Ein Player sollte außerdem so flexibel sein, gegen Unstimmigkeiten in eben dieser Reihenfolge unempfindlich zu sein, damit die Zulassung von Akkorden in dieser Hinsicht keine zusätztlichen Probleme aufwirft.

Das folgende Diagramm veranschaulicht die Weise, in der Akkordtöne von einem monophonen Playerprogramm aussortiert und damit entfallende Notenbindungen aufgelöst werden müssen.

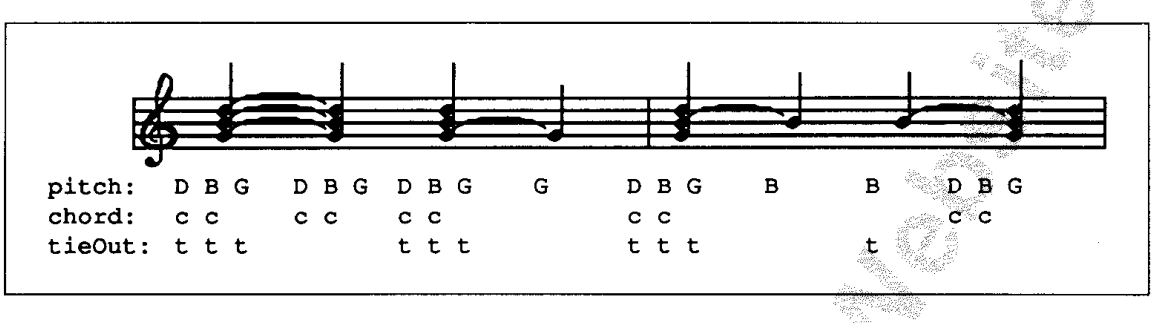


Abb. 2.2: Akkordauflösung

Die Länge des Events »Pause« (mit dem *sID*-Wert *SID_Rest*, 128) richtet sich ebenso wie die Notenevents nach den Elementen *division*, *dot* und *nTuplet*. In diesem Fall sind die Elemente *chord* und *tieOut* ohne Bedeutung und müssen auf 0 gesetzt sein.

Sollten Sie, wie gesagt, einen C-Compiler oder eine andere Sprache verwenden, die die in der *SEvent*-Strukur auftauchenden Bitfelder nicht unterstützt, so können sie statt dessen die folgenden Konstanten einsetzen:

```
#define noteChord    (1<<7)              /* Akkordnote */

#define noteTieOut   (1<<6)              /* Notebindung */

#define noteNShift   4                   /* Verschiebungswert f. NTuplet */
#define noteN3       (1<<NoteNShift)     /* Triole */
#define noteN5       (2<<NoteNShift)     /* Quintole */
#define noteN7       (3<<NoteNShift)     /* Septole */

#define noteDot      (1<<3)              /* Punktierung */

#define noteD1       0                   /* ganze Note */
#define noteD2       1                   /* halbe Note */
#define noteD4       2                   /* Viertelnote */
#define noteD8       3                   /* Achtelnote */
#define noteD16      4                   /* 16tel-Note */
#define noteD32      5                   /* 32tel-Note */
#define noteD64      6                   /* 64tel-Note */
#define noteD128     7                   /* 128tel-Note */

#define noteDMask    noteD128            /* Bitmaske f. division */
#define noteDurMask  0x3F                /* Bitmaske f. Gesamtdauer */
```

Die technischen Events beziehen sich ohne Ausnahme auf Zustände des Playerprogrammes, insbesondere der einzelnen Tracks, die wiedergegeben werden. Diese Klasse von Events wird in den folgenden Kapiteln besprochen.

Das Event »Instrumentänderung«

Einer der Zustände, dem die Wiedergabe eines Tracks unterworfen ist, ist die Einstellung des Instruments. Jedes *SNote*-Event innerhalb eines Tracks bezieht sich implizit auf eine Vorauswahl

eines solchen Instrumentes, so daß diese Angabe nicht bei der Verarbeitung jedes einzelnen Events erfolgen muß. Das *SEvent*, das die Änderung dieses Zustandes bewirkt, hat den *sID*-Wert *SID_Instrument* und im *data*-Element steht die Registernummer des INS1-Chunks, in dem das gewünschte Instrument definiert wird.

Immer wenn von einem Zustand oder Modus die Rede ist, in dem die Noten eines Tracks abgespielt werden, stellt sich die Frage nach dem Anfangszustand, also dem Zustand bevor irgendein diesen Zustand reglementierendes Event aufgetreten ist. Der SMUS-Standard schlägt vor, daß ein Playerprogramm zu Beginn der Wiedergabe jedem Track seine eigene Tracknummer als Instrument-Registernummer zugeordnet werden soll. Dies birgt schon allein die Schwierigkeit, daß nicht in jedem Score unbedingt mindestens ebenso viele Instrumente wie Tracks definiert sein müssen. Deshalb bedienen sich die meisten Player eines sog. Default-Instruments, welchem jeder Track als Instrumentreferenz beim Beginn der Wiedergabe zugeordnet wird.

Das Event »Taktmaß setzen«

Das *SNote*-Event mit dem *sID*-Wert *SID_TimeSig* bestimmt das Taktmaß, auf das sich die Noten eines Tracks beziehen. Dieser Wert gibt die Anzahl der Taktschläge in einem Takt und deren Einheit an. Dieses Event hat lediglich eine Funktion innerhalb eines Score-Editors, da die effektive zeitliche Abfolge der Noten eines Tracks hiervon nicht berührt wird. Weniger professionelle Score-Editoren lassen nur ein Taktmaß innerhalb des gesamten Scores zu, höherentwickelte erlauben die Änderung des Taktmaßes im Score. Es finden sich jedoch praktisch keine Editoren, die verschiedene Taktmaße der einzelnen Tracks zu gleicher Zeit unterstützen, wohl weil diese Option in der realen Kompositionstechnik zu selten benötigt wird und auch einen zu hohen Anspruch an die Notendarstellung des jeweiligen Programms stellen würde.

Die C-Definition für ein *SID_TimeSig*-Event lautet:

```
typedef struct {
   UBYTE type;              /* ::= SID_TimeSig */
   unsigned TimeNSig 5:,    /* Taktmaß Zähler */
   TimeDSig 3:;             /* Taktmaß Nenner */
} STimeSig;
```

Anmerkung zur Implementation: Wie bereits erwähnt, können Sie entweder diese Definition verwenden, oder müssen durch entsprechende logische Verknüpfungen die gewünschten Werte gewinnen, abhängig von der von Ihnen benützten Programmiersprache. Die dazu notwendigen Konstanten finden Sie am Ende dieses Kapitels.

Das Strukturelement *type* enthält, wie in allen *SEvents*, die dem Event-Typ zugehörige *sID*, in diesem Fall *SID_TimeSig*.

Das Bitfeld *TimeNSig* bestimmt den Zähler (engl.: Numerator) des Taktmaßes, also die Anzahl der Taktschläge (Beats) eines Taktes. Zu dem Wert dieses Feldes muß 1 addiert werden, um den effektiven Zählerwert zu erhalten. Bei einem Wertebereich von 5 Bit (von 0 bis 31) ergibt sich dadurch eine maximale Anzahl von 32 Beats pro Takt.

Das Bitfeld *TimeDSig* bestimmt den Nenner (engl.: Denominator) des Taktmaßes.

Da dieses Event lediglich für einen Score-Editor von Bedeutung ist, ist dessen Angabe für ein reines Playerprogramm nicht zwingend erforderlich. Zur visuellen Darstellung eines Musikstückes sind jedoch Taktstriche von großem Nutzen, um ein Musikstück leichter bearbeitbar zu machen. Deshalb sollte jede SMUS-FORM am Beginn einer ihrer Tracks ein solches Event enthalten. Ist dies aus irgendwelchen Gründen nicht der Fall, so sieht der SMUS-Standard vor, beim Lesen eines Score-Files das Taktmaß $^4/_4$ vorauszusetzen.

Die C-Definitionen für die hier benötigten Konstanten lauten:

```
#define timeNMask    0xF8    /* Bitmaske f. TimeNSig */
#define timeNShift   3       /* Verschiebungswert f. TimeNSig */
#define timeDMask    0x07    /* Bitmaske f. TimeDSig */
```

Das Event »Tonart setzen«

Ebenfalls nur für einen Score-Editor interessant ist *SID_KeySig*. Dieses Event bestimmt die Tonart, in der der Score dargestellt wird, und damit die Vorzeichen. Der Wert des *data*-Elements hat hier folgende Bedeutung:

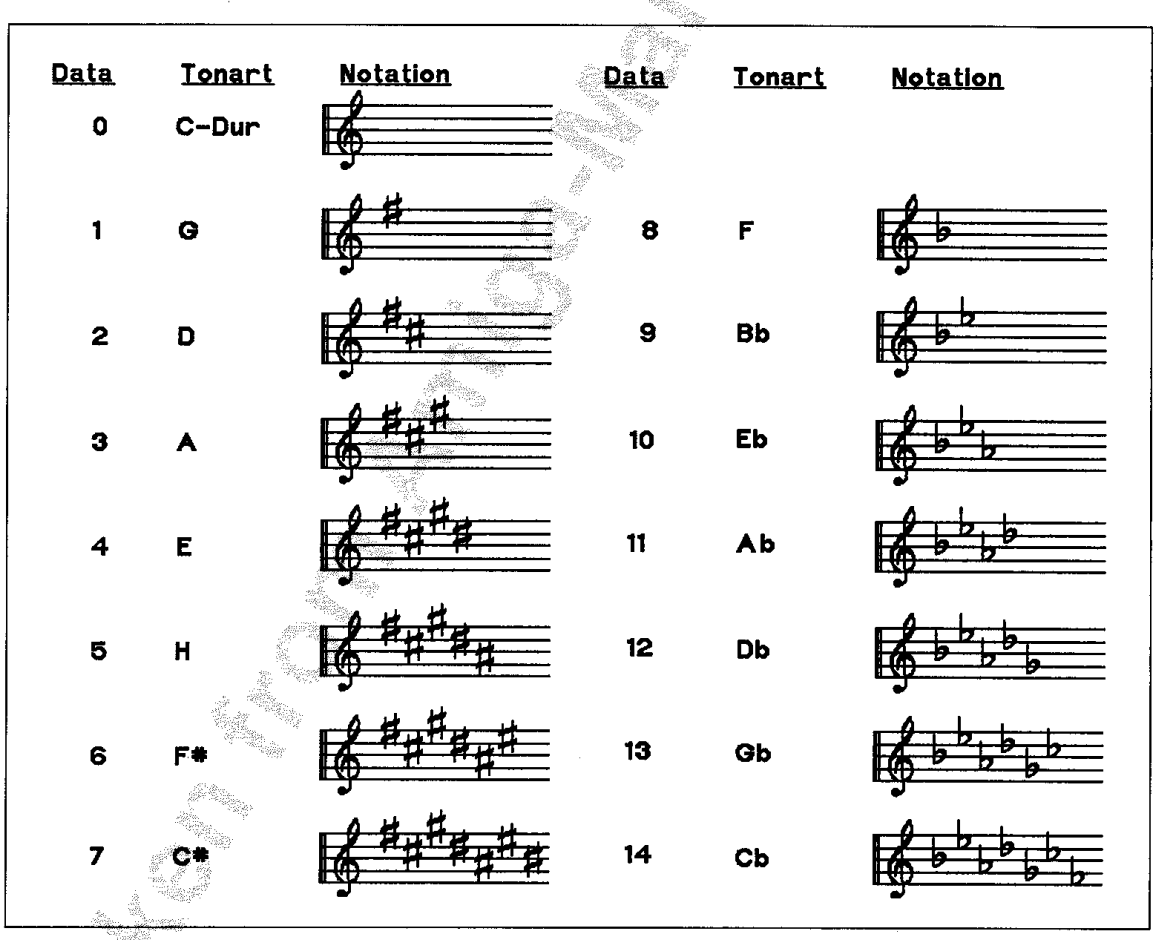


Abb. 2.3: Keysigs

Taken from the Amiga-Mandis-Wiki

Analog zum Taktmaß gilt dieses Event meist für sämtliche Tracks des Scores. Fehlt dieses Event, so wird normalerweise C-Dur als Default-Tonart verwendet.

Das Event »Notenschlüssel setzen«

Mit Hilfe des Event *SID_Clef* läßt sich der Notenschlüssel bestimmen, in dem der jeweilige Track dargestellt wird. Auch dieses Event hat nur für einen Score-Editor eine Funktion. Die *data*-Werte 0–3 stehen für Violin-, Baß-, Alt-, und Tenorschlüssel.

Das Event »Lautstärke setzen«

Das Event *SID_Dynamic* bestimmt die individuelle Lautstärke eines Tracks. Hiermit lassen sich die Lautstärkeverhältnisse zwischen den einzelnen Stimmen regeln (Mixing). Dieser Wert muß noch mit dem im *SHDR*-Chunk angegebenen *volume*-Wert skaliert werden, um die effektive Lautstärke zu erhalten.

Der Wertebereich des *data*-Elementes umfaßt 0 bis einschließlich 127, wobei 0 unendlich leise (–[unendlich] dB) und 127 Vollausschlag (0 dB) bedeutet. Fehlt dieses Event am Beginn eines Tracks, so wird als Default-Wert 127 angenommen.

Das Event »Tempo setzen«

Das Event *SID_Tempo* erlaubt die Änderung des Wiedergabetempos des Scores, es gilt also für alle Stimmen gleichzeitig. Da das *data*-Element leider nur 1 Byte groß ist, kann die Angabe nicht in 128tel Schlägen pro Minute erfolgen (wie im SHDR-Chunk) sondern nur in Schlägen/Minute.

Die Events »MIDI-Kanalnummer setzen« und »MIDI-Preset setzen«

Für Playerprogramme, die angeschlossene MIDI-Hardware unterstützen, oder auch Programme, die ankommende MIDI-Daten ins SMUS-Format umwandeln (sog. Recordingprogramme), sind die Events *SID_MIDI_Chnl* und *SID_MIDI_Preset* gedacht.

Das *SID_MIDI_Chnl*-Event enthält als *data*-Element die Nummer des MIDI-Kanals, dem der jeweilige Track zugeordnet werden soll. Die Abarbeitung dieses Events bewirkt, daß sämtliche Noten fortan an die genannte Kanalnummer übergeben werden, sobald dieser Track durch ein *SID_Instrument*-Event auf ein MIDI-Instrument eingestellt wird (d.h. einen *INS1*-Chunk mit *type = INS1_MIDI*).

Das *SID_MIDI_Preset*-Event enthält als *data*-Element eine MIDI-Preset-Nummer, die dem durch *SID_MIDI_Chnl* angegebenen Kanal eine bestimmte Voreinstellung innerhalb des angeschlossenen Gerätes zuordnet. Damit kann diesem Kanal, abhängig von den Fähigkeiten des externen Gerätes, ein beliebiges Instrument gegeben werden.

Taken from Manuel-Website

2.3 Programmspezifische (Non-Standard-) Chunks

Im folgenden Kapitel beschäftigt sich mit den Chunk-Typen, die von den einzelnen Score-Editoren und Kompositionsprogrammen erzeugt werden, um ihren Dateien programmspezifische Daten und Einstellungswerte hinzuzufügen. Es handelt sich hierbei ausschließlich um Property-Chunks, also Eigenschaftsangaben, deren Gültigkeit sich über die gesamte FORM-Struktur erstreckt, in die sie eingebettet sind.

Sonix: Der SNX1-Chunk

Sonix verwendet zur Speicherung von diversen Schiebereglern und Gadgets einen Chunk, der eine sog. SNX1-Struktur enthält. Diese ist wie folgt definiert:

```
struct SNX1 {
   UWORD s1_Transpose;
   UWORD s1_Tune;
   UWORD s1_Keyboard;
   UWORD s1_Mask;
   ULONG s1_Select[8];
   BYTE s1_KeyAssign[128];
};
```

s1_Transpose: Bezeichnet die Anzahl der Halbtöne, um die der Score bei der Wiedergabe transponiert wird (Transpose-Regler). Wertebereich: 0 – 255; 128 = keine Transponierung, jede Veränderung um einen +-16-Schritt bedeutet Transponierung um einen Halbton.

s1_Tune: Hier wird die Feinstimmung für die Wiedergabe angegeben (Tune-Regler). Wertebereich: 0 – 255; 128 = Standardstimmung (a1 = 440 Hz), 0 = $-\frac{1}{2}$ Halbton, 255 = $+\frac{1}{2}$ Halbton.

s1_Keyboard: Dieses Feld gibt die Stellung des Keyboard-Gadgets und damit die Nummer des Kanals an, über den die Computer-Tastatur wiedergegeben werden soll. Wertebereich: 0 – 7.

s1_Mask: Zur Zeit unbenutzt.

s1_Select: Hier ist der Zustand der acht Track-Schalter enthalten: 0 = stummgeschaltet, 1 = volle Lautstärke, 2 = halbe Lautstärke

s1_KeyAssign: In dieser Tabelle stehen die MIDI-Tonnummern, die den Rawkey-Werten der einzelnen Tasten des Computer-Keyboards zugeordnet werden. Wertebereich: –1 = keine Zuordnung, 0–127 MIDI-Tonnummer.

2.4. SMUS-Schnellübersicht

2.4.1 Typ-Definitionen

Hier eine Zusammenfassung sämtlicher C-Definitionen für die SMUS-FORM. Strukturelemente werden in der beschriebenen Reihenfolge aneinandergereiht. Ein UBYTE-Feld wird zu einem 8-Bit-Byte zusammengefaßt. Programme sollten alle Pad-Elemente auf 0 setzen.

```
define ID_SMUS MakeID('S','M','U','S')
define ID_SHDR MakeID('S','H','D','R')

typedef struct {
   UWORD tempo;      /* Tempo in (Viertelnoten/Minute) * 128 */
   UBYTE volume;     /* Gesamtlautstärke von 0-127 */
   UBYTE ctTrack;    /* Anzahl der "TRAK"-Chunks */
} SScoreHeader;

#define ID_Name MakeID('N','A','M','E')
/* enthält einen char[]-Vektor (den Namen des Scores) */

#define ID_CopyRight MakeID('(','c',')',' ')
/* enthält einen char[]-Vektor (Copyright-Hinweis) */

#define ID_AUTH MakeID('A','U','T','H')
/* enthält einen char[]-Vektor (den Namen des Autors etc.) */

#define ID_ANNO MakeID('A','N','N','O')
/* enthält einen char[]-Vektor (Erstellungsdatum o.Ä.) */

#define INS1_Name  0
#define INS1_MIDI  1

typedef struct {
   UBYTE register;      /* Instr.-Registernummer */
   UBYTE type;          /* Typ: Name oder MIDI */
   UBYTE data1,data2;   /* abhängig von type */
   CHAR  name[];        /* Instr.-Name */
} RefInstrument;


#define ID_TRAK MakeID('T','R','A','K')

/* Definition von SEvent */
typedef struct {
   UBYTE sID;
   UBYTE data;
} SEvent;

/* mögliche sID-Typen */
#define SID_FirstNote      0
#define SID_LastNote     127

/* sID`s im Bereich von SID_FirstNote bis
SID_LastNote stellen Notenwerte dar. */

#define SID_Rest         128
#define SID_Instrument   129
#define SID_TimeSig      130
#define SID_KeySig       131
#define SID_Dynamic      132
#define SID_MIDI_Chnl    133
```

Taken from Amiga-Manuals-Website

```
#define SID_MIDI_Preset    134
#define SID_Clef           135
#define SID_Tempo          136

/* Die sID-Werte von 144 bis 159 sind für
Instant-Music SEvents reserviert (!!!),
die restlichen Werte bis 254 für zukünftige
Erweiterungen des Standards. */

#define SID_Mark    255
/* Definiert als Ende-Zeichen für
Repräsentationen der Daten im Arbeitsspeicher */


typedef struct {
   UBYTE    tone;        /* MIDI-Tonnummer von 0 bis 127 */
   unsigned chord    :1, /* Akkord? */
            tieOut   :1, /* 1 bedeutet gebundene Note */
            nTuplet  :2, /* 0 = normale Note, 1 = Triole,
                         /* 2 = Quintole, 3 = Septole */
            dot      :1, /* punktierte Note? */
            division :3; /* 0 = ganze Note, 1 = halbe Note,
                            2 = Viertelnote, 7 = 128tel Note */
} SNote;


#define noteChord   (1<<7) /* Akkordnote */

#define noteTieOut  (1<<6) /* Notebindung */

#define noteNShift  4      /* Verschiebungswert f. NTuplet */
#define noteN3             (1<<NoteNShift)     /* Triole */
#define noteN5             (2<<NoteNShift)     /* Quintole */
#define noteN7             (3<<NoteNShift)     /* Septole */

#define noteDot            (1<<3) /* Punktierung */

#define noteD1             0      /* ganze Note */
#define noteD2             1      /* halbe Note */
#define noteD4             2      /* Viertelnote */
#define noteD8             3      /* Achtelnote */
#define noteD16            4      /* 16tel-Note */
#define noteD32            5      /* 32tel-Note */
#define noteD64            6      /* 64tel-Note */
#define noteD128    7      /* 128tel-Note */

#define noteDMask   noteD128 /* Bitmaske f. division */
#define noteDurMask 0x3F   /* Bitmaske f. Gesamtdauer */
```

Taken from Amiga-Manuals-Website

```
typedef struct {
   UBYTE type;                /* ::= SID_TimeSig */
   unsigned TimeNSig 5:,      /* Taktmaß Zähler */
   TimeDSig 3:;               /* Taktmaß Nenner */
} STimeSig;


#define timeNMask   0xF8      /* Bitmaske f. TimeNSig */
#define timeNShift  3         /* Verschiebungswert f. TimeNSig */
#define timeDMask   0x07      /* Bitmaske f. TimeDSig */
```

2.4.2 SMUS-Standardausdrücke

Hier eine Liste der Standardausdrucke der SMUS FORM Syntax. Eine SMUS FORM kann ein selbständiges IFF-File oder ein Teil eines solchen sein.

```
SMUS      ::= "FORM" #{ "SMUS" SHDR [NAME] [Copyright] [AUTH] [IRev]
              ANNO* INS1* TRAK* InstrForm* }

SHDR      ::= "SHDR" #{ SScoreHeader  }
NAME      ::= "NAME" #{ CHAR*       } [0]
Copyright ::= "(c) " #{ CHAR*       } [0]
AUTH      ::= "AUTH" #{ CHAR*       } [0]
IRev      ::= "IRev" #{ ...         }

ANNO      ::= "ANNO" #{ CHAR*       } [0]
INS1      ::= "INS1" #{ RefInstrument  } [0]

TRAK      ::= "TRAK" #{ SEvent*     }
InstrForm ::= "FORM" #{ ...         }
```

Das Zeichen "#" steht für das *ckSize*-Feld (LONG, 32 Bits!), also die Länge der Daten innerhalb der geschweiften Klammern { }. Buchstaben wie z.B. IDs werden in Anführungszeichen " " gesetzt, was in eckigen Klammern [ ] steht ist optional, und "*" bedeutet 0, eine oder mehrere Wiederholungen des Ausdrucks davor. Ein manchmal benötigtes Pad-Byte wird immer als [0] angegeben.

Die Reihenfolge der Chunks muß im SMUS-Format nicht so streng an der obigen Standardschreibweise ausgerichtet sein. Die Chunks SHDR, Name, Copyright, AUTH, IRev, ANNO und INS1 können in beliebiger Reihenfolge auftreten, solange sie noch vor den TRAK-Chunks sind. Der Chunk »InstrForm« repräsentiert irgendein in die SMUS FORM eingebettetes Instrumentenformat. Dies könnte z.B. das 8SVX IFF – Format sein. Natürlich müssen nicht erkannte Instrumentenfomate ignoriert werden.

Taken from Amiga-Manuals-Website

Instrumentenformate 3

3.1 Das 8SVX-IFF-Format (8 Bit Sampled Voice)

Zur Speicherung der beim Amiga üblichen 8-Bit-Samples ist im IFF-Format ein eigener FORM-Typ vorgesehen: Das 8SVX-(8-Bit-Sampled-Voice-)Format. In diesem Format können sowohl ganze Klangstücke mit fester Samplingrate (z.B. für unveränderliche Effekte wie kleine Sprachsequenzen) als auch Musikinstrumente mit variabler Tonhöhe und Spieldauer definiert werden, wobei für jede Oktave eine eigene Wellenform existiert. Bei den Musikinstrumenten lassen sich grundsätzlich zwei Typen unterscheiden: Instrumente mit einer konstanten Haltephase (d.h. die Tondauer richtet sich nach dem Drücken einer Taste oder dem Notenwert in einem Score), und solche, die nach einer gewissen Zeit verklingen. Letztere lassen sich mit einem einzigen Datenfeld darstellen, das nur einmal abgespielt wird (OneShot-Part). Diese Form ist geeignet, ein Instrument ohne konstante Energiezufuhr zu imitieren, wie z.B. eine Trommel, eine Gitarren- oder Klaviersaite etc. Erstere hingegen enthalten einen nur einmal abgespielten Teil, und einen, der ständig wiederholt wird, solange die Taste gedrückt bleibt (OneShot- und RepeatPart). Hiermit lassen sich die meisten Instrumente mit einer Haltephase befriedigend darstellen, wie z.B. eine Flöte, eine Orgel oder die menschliche Stimme.

Die Darstellung erfolgt, gemäß der Audio-Hardware des Amiga, im 8-Bit-Zweierkomplement. Um ein optimales Signal-Rausch-Verhältnis zu erzielen, müssen die verwendeten Samples möglichst den maximalen Aussteuerungsbereich verwenden (–128 bis 127). Die Lautstärke des Klangstückes oder Instruments sollte deshalb nur über das Lautstärkeregister verändert werden. Hierbei ist im 8SVX-Format sowohl eine Angabe für die Gesamtlautstärke vorgesehen als auch eine Möglichkeit zur Gestaltung einer Hüllkurve.

Bei der Verwendung als Musikinstrument ist für jede Oktave ein Datenfeld vorgesehen, wobei nur die Größe des Datenfeldes der höchsten Oktave angegeben ist, welches natürlich auch aus einer geraden Anzahl von Samples bestehen muß. Die Felder der tieferen Oktaven müssen deshalb jeweils die 2^n-fache Länge des höchsten aufweisen, für die unter der höchsten also die doppelte, für die darunter die vierfache usw. Dies schafft zwei Probleme: Erstens klingen Töne, die genau zwischen den gesampleten Tonhöhen liegen, meist ziemlich unnatürlich, da sie entweder ums $1^1/_2$-fache tiefer oder höher abgespielt werden als sie aufgenommen wurden. Zweitens ist man beim Entwurf der Sampledaten ziemlich eingeengt: Hat man z.B. einen natürlichen Klang digitalisiert und die entsprechenden OneShot- und Repeatpunkte bestimmt, so ist man für alle darunter- und darüberliegenden Oktaven an diese Maße gebunden, da deren Datenfelder ja jeweils

genau doppelt, vierfach bzw. halb oder ein Viertel so groß sein müssen. Um diese Schwierigkeiten zu umgehen, müßte man sich etwa einer LIST-Anordnung von einzelnen FORM 8SVX bedienen, um mehr Wellenformen pro Oktave definieren zu können.

3.1.1 Standard-8SVX-Chunks

In diesem Kapitel sollen die Chunk-Typen, die die Syntax der 8SVX-FORM bilden, im einzelnen erläutert werden. Eine 8SVX-FORM besteht aus dem (notwendigen) VHDR-Chunk und den (optionalen) Chunks NAME, AUTH und ANNO sowie ATAK und RLSE. Daran schließt sich der BODY-Chunk an, der die Wellenformen sämtlicher Oktaven enthält. Weitere 8SVX-Chunks könnten in Zukunft definiert werden, wie z.B. eine Fourier-Darstellung der Wellenform.

'FORM'	464F524D	00009B90
'8SVX'	38535658	
'VHDR'	56484452	00000014
oneShot, Repeat	00000480	00000080
sPHiCycle, sPSec	00000008	3101
ctOct, sComp., vol.	05 00	00010000
'NAME'	4E414D45	00000005
'Piano'	50 69 61 6E 6F 00	
'ATAK'	4154414b	00000012
	0021	00010000
	0042	0000CCCC
	1568	00005555
'BODY'	424F4459	00009B00
	00 00 FF FB FA	
	...	

Abb. 3.1: Piano

Taken from Amiga Magazin Archive

Abb. 3.1 zeigt eine 8SVX-FORM mit dem Namen »Piano«, deren Anschlagsverhalten durch einen ATAK-Chunk definiert wird.

Der VHDR-Chunk

Der Property-Chunk VHDR enthält eine Voice8Header-Struktur, wie sie im folgenden beschrieben wird. In dieser Struktur sind die Angaben und Wiedergabeparameter für die Wellenformdaten im BODY-Chunk enthalten (zur Speicherverteilung dieser Wellenformen siehe 3.1.2.4 »Der BODY-Chunk«):

```
#define ID_8SVX      MakeID('8', 'S', 'V', 'X')
#define ID_VHDR      MakeID('V', 'H', 'D', 'R')

/* ---------- Voice8Header ----------------------------------------*/
typedef LONG Fixed;          /* Festkommawert mit 16 Bit links und 16 Bit
                              * rechts vom Komma. Kann gesehen werden als
                              * eine Zahl aus 2^16teln, also 65536teln. */
#define Unity 0x10000L       /* Unity = Fixed 1.0 = maximale Lautstärke */

/* sCompression: gewählter Kompressions-Algorithmus */
#define sCmpNone       0     /* nicht komprimiert */
#define sCmpFibDelta   1     /* Fibonacci-delta-Kodierung */
                             /* Weitere Kompressionsarten möglich */
typedef struct {
   ULONG oneShotHiSamples,   /* # Samples im OneShotPart (höchste Okt.)*/
         repeatHiSamples,    /* # Samples im RepeatPart (höchste Okt.)*/
         samplesPerHiCycle;  /* # Samples/Cycle (höchste Okt),sonst 0 */
   UWORD samplesPerSec;      /* Sampling-Rate */
   UBYTE ctOctave,           /* Anzahl der Oktaven im BODY-Chunk */
         sCompression;       /* gewählter Kompressions-Algorithmus */
   Fixed volume;             /* Lautstärke von 0 bis Unity (1.0, volle Lautst.) */
} Voice8Header;
```

Jede 8SVX-FORM kann sowohl einen OneShot-Teil, der nur einmal gespielt wird, als auch einen Repeat-Teil enthalten, der im Anschluß an den OneShot-Teil ständig wiederholt wird. Die Elemente *oneShotHiSamples* und *repeatHiSamples* enthalten die Längen dieser Teile in der höchsten (also vom Speicherbedarf her kleinsten) Oktave der Wellenform. In den nächst unteren Oktaven sind die beiden Teile jeweils doppelt, vierfach, achtfach etc. so groß. Einer von beiden Werten kann auch 0 sein, so daß in allen Oktaven nur ein OneShot- oder ein Repeat-Teil vorhanden ist.

Die meiste Abspielsoftware unterliegt hinsichtlich dieser Größen einigen Beschränkungen: Alle OneShot- und Repeat-Teile im BODY-Chunk müssen eine gerade Anzahl von Bytes lang sein; manche Software kann als Wellenformlängen sogar nur Potenzen von 2 verarbeiten. Eine 8SVX-Leseroutine sollte jedoch zumindest einen ungeraden BODY-Chunk korrigieren können und eine ungerade unterste Oktave oder ungerade Wellenformen überhaupt ignorieren.

Das Element *samplesPerHiCycle* enthält die Anzahl der Samples pro Wellenformzyklus in der höchsten Oktave, oder bei einem digitalisierten natürlichen Instrument pro Schwingungszeit T des Grundtones (sofern vorhanden). Diese Angabe ist nur bei einem Musikinstrument erforderlich

und dient zum Errechnen der Ausgaberate für eine bestimmte Tonhöhe. Jede Oktave unterhalb der höchsten sollte jeweils doppelt, vierfach etc. so viele Samples/Cycle enthalten. (Der effektive Tonabstand betrüge ja sonst nicht genau 1 Oktave.)

samplesPerHiCycle kann oft nur ein Durchschnittswert sein, z.B. wenn der OneShot-Part bereits selbst eine Tonhöhenänderung enthält. Dann sollte *samplesPerHiCycle* am Wert für den Repeat-Part ausgerichtet werden. Da zur Vermeidung von abrupten Spannungsänderungen ("Clicks") Repeat-Teile stehts auf ungefähr dem gleichen Wert beginnen und enden müssen, sollte das Verhältnis *repeatHiSamples/samplesPerHiCycle* immer ganzzahlig sein. Viele Programme nehmen *samplesPerHiCycle* als Potenz von 2 an, teilweise sogar ohne Prüfung weiterer Bits. Das führt in dem Fall, daß hier ein anderer Wert steht, zu einer um eine Oktave zu hohen oder zu tiefen Wiedergabe.

Das Element *samplesPerSec* bestimmt die Sampling-Rate, mit der die Wellenform im BODY-Chunk wiedergegeben werden soll. Dieses wird nur bei Klangsequenzen benötigt, die mit einer festen Datenrate abgespielt werden sollen, bei digitalisierten Klängen üblicherweise mit der Aufnahmerate. Programme, die eine 8SVX-FORM als Instrument verwenden wollen, ignorieren diese Angabe und wählen die aus der erwünschten Tonhöhe und *samplesPerHiCycle* errechnete Sampling-Rate.

Das Element *ctOctave* enthält die Anzahl von Oktaven, deren Wellenformen im BODY-Chunk enthalten sind. Hieraus läßt sich zusammen mit *oneShotHiSamples* und *repeatHiSamples* die Länge des BODY-Chunks bzw. die Position der Daten einer bestimmten Oktave bestimmen (siehe BODY-Chunk).

Das Element *sCompression* gibt den Kompressionsalgorithmus an, der die Daten im BODY-Chunk unterzogen wurden. Hier sollte einer von den oben definierten Werten stehen (im Normalfall 0 = keine Kompression). Bevor die Daten im BODY-Chunk abgespielt werden können, muß man diesen natürlich zuerst mit dem entsprechenden Dekompressionsalgorithmus behandeln. Ein BODY-Chunk wird immer als Ganzes kompromiert oder dekomprimiert. Alle Angaben im VHDR-Chunk beziehen sich auf den nicht komprimierten Zustand. Der Fibonacci-Delta-Algorithmus wird in Abschnitt 3.1.1.5 behandelt.

Das Element *volume* bestimmt die Lautstärke der Klangsequenz oder des Instrumentes. Dies erlaubt die Ausnutzung des vollen Aussteuerungsbereichs von –128 bis 127. Dieser Wert sollte z.B. von einem Playerprogramm noch mit der internen Gesamtlautstärke und ggf. mit einer Hüllkurve skaliert werden (siehe ATAK- und RLSE-Chunks).

Wie erwähnt, läßt sich eine 8SVX-FORM grundsätzlich auf zwei verschiedene Arten verwenden: Als feste Klangsequenz (Sampling-Rate ist vorgegeben) oder als Musikinstrument (Sampling-Rate wird frei bestimmt). Um aus einer gegebenen Wellenform das eine oder andere zu machen, geht man folgendermaßen vor:

Erstellung einer festen Klangsequenz: Da eine solche meistens nur aus einem OneShot-Part besteht, sollten *oneShotHiSamples* die Länge des OneShot-Parts und *repeatHiSamples* eine 0 enthalten (ein Repeat-Teil ist natürlich ggf. auch zugelassen). *samplesPerHiCycle* sollte gleich 0, *samplesPerSec* gleich der Samplingrate und *ctOctave* gleich 1 sein. Da die Lautstärke durch

volume bestimmt wird, sollte bei den Wellenformdaten der volle Aussteuerungsbereich von –128 bis 127 genutzt werden. Experimentieren Sie mit Kompressionsalgorithmen, da diese eine erfreuliche Verkürzung der Ladezeiten von Diskette bewirken können. Wenn der dekomprimierte Klang befriedigend ist, vermerken Sie die verwendete Kompression in *sCompression*.

Erstellung eines Musikinstruments: Um eine 8SVX-FORM als Musikinstrument einzusetzen, digitalisieren oder errechnen Sie zuerst so viele Oktaven, wie Sie zur Wiedergabe zur Verfügung haben wollen, und schreiben deren Anzahl in *ctOctave*. Abhängig vom Instrumententyp können entweder *oneShotHiSamples* oder *repeatHiSamples* auch gleich 0 sein. Optimieren Sie den OneShot- und vor allem den Repeat-Part einer bestimmten Oktave (am besten der höchsten), bis er gut klingt. Aus der Frequenz der Grundschwingung und der Länge der Wellenform der höchsten Oktave errechnen Sie *samplesPerHiCycle*.

Dann entnehmen Sie den anderen Oktaven für deren OneShot- und Repeat-Parts jeweils entsprechend große Teile, wobei das Finden von optimalen Wiederholungspunkten (Looping) nun leider ziemlich erschwert ist, da man in der Länge für die beiden Teile nun bereits festgelegt ist. Dies ist bei errechneten Daten meist noch keine große Schwierigkeit, bei digitalisierten Wellenformen kann dies jedoch zum Geduldsspiel werden (siehe Kapitel 5.2). Man kann dies vermeiden, indem man nur eine einzige Oktave digitalisiert, und die restlichen Oktaven durch Verdoppelung und Extrapolation (d.h. man bildet Zwischenwerte für die einzelnen Samples) oder Interpolation (hier indem man jedes zweite Sample entfernt). Um das Verhältnis zwischen OneShot- und Repeat-Part den Eigenschaften des Instrumentes anzupassen, kann der Beginn des OneShot-Parts auch mit Nullen aufgefüllt werden, was in gewissen Grenzen keine merklichen Effekte auf das Anschlagverhalten des Tones hat.

Auch hier sollte in jedem Fall der volle Aussteuerungsbereich genutzt und jegliche Wellenformmodulation über das Lautstärkeregister durchgeführt werden. Gibt man zusätzlich einen Wert für *samplesPerSec* an, so kann das Instrument auch als »feste Klangsequenz« genutzt werden. Von einem verzerrungsträchtigen Kompressionsalgorithmus wie der Fibonacci-Delta-Kodierung muß bei der Erstellung von Instrumenten unbedingt abgeraten werden.

Merke: Im Prinzip kann jede Instrument- 8SVX-FORM auch als »feste Klangsequenz« verwendet werden und umgekehrt. Wichtig ist, daß ein Musikinstrument unbedingt einen Wert für *samplesPerHiCycle* enthalten muß und eine feste Klangsequenz einen Wert für *samplesPerSec*.

Um eine 8SVX-FORM in beiden Variationen wiederzugeben, sind jeweils die folgenden Schritte notwendig:

Abspielen einer festen Klangsequenz: Um eine 8SVX-FORM als feste Klangsequenz wiederzugeben, muß zunächst eine der Oktaven ausgewählt werden, falls *ctOctave* größer als 1 ist. Normalerweise wird man hier die tiefste Oktave wählen, da sie die größte Auflösung aufweist. Zuerst erfolgt die Ausgabe des OneShot-Parts, dann wird der Repeat-Part in ständiger Wiederholung ausgegeben (siehe Kapitel 2 und 3). Die Lautstärke wird aus *volume* errechnet, evtl. unter Berücksichtigung einer Hüllkurve (siehe ATAK- und RLSE-Chunk).

Abspielen eines Musikinstrumentes: Um eine 8SVX-FORM als Instrument wiederzugeben, muß zuerst die nächstliegende verfügbare Oktave bestimmt werden. Sollte sich die gewünschte Frequenz um mehr als eine Oktave von der nächsten erreichbaren unterscheiden (zu hoch oder zu

tief), so muß entweder eine passende Wellenform aus den vorhandenen Daten errechnet oder die Abspielanforderung ignoriert werden. Die Wiedergaberate sollte analog der Formel

Wiedergaberate = Frequenz * samplesPerHiCycle * Oktave#

Formel 50

gewählt werden. Wenn *samplesPerHiCycle* als Potenz von 2 angenommen wird, kann ein Abspielprogramm vorberechnete Tabellen von Wiedergaberaten verwenden.

Die optionalen Text-Chunks NAME, (c), AUTH, ANNO

In der 8SVX-FORM gibt es eine Klasse von Chunks, die zusätzliche Informationen über das Intrument wie Autor, Erstellungsdatum oder evtl. Urheberrechtsschutz enthalten können. Die Datenfelder dieser Chunks bestehen aus den im IFF-Standard üblichen Charakter-Werten aus dem Bereich " " (Space, hex $20) bis "~" (Tilde, hex $7F). Die Größe *ckSize* gibt hier die Anzahl der Bytes wieder, die von dem jeweiligen Text-String belegt wird.

Der NAME-Chunk enthält den Namen des entsprechenden Instruments. Man beachte, daß sich dieser Name keineswegs mit dem Dateinamen des jeweiligen Instruments decken muß, wobei der im NAME-Chunk angegebene Name den »eigentlichen« Namen des Instruments darstellt. Am besten ist es also, beim Entwurf eines Instrument-Editors darauf zu achten, daß sich diese beiden Angaben niemals widersprechen können.

Für die übrigen Chunks ("(c) ", "ANNO") gilt das bereits unter Kapitel 2.2.2 erwähnte.

Die optionalen Chunks ATAK und RLSE

Mit Hilfe der ATAK- und RLSE-Chunks ist es möglich, für einen Sound in der 8SVX FORM eine stückweise lineare Amplitudenmodulation oder Hüllkurve anzugeben. In diesen Chunks befindet sich jeweils eine Tabelle mit Lautstärkewerten, mit denen sich das Anschlag und Ausschwingverhalten des Sounds beschreiben läßt. Ein Abspielprogramm kann diese mit der Lautstärke in der *Voice8Header*-Struktur multiplizieren, oder sie auch ignorieren und seine eigenen Einstellungen verwenden.

Die C-Definitionen lauten:

```
#define ID_ATAK      MakeID('A', 'T', 'A', 'K')
#define ID_RLSE      MakeID('R', 'L', 'S', 'E')

/* ---------- Envelope ATAK & RLSE -------------------------------------*/
typedef struct {
    UWORD duration; /* Dauer d. Hüllkurvenabschnitts in ms, > 0 */
    Fixed dest;            /* Lautstärkefaktor am Ende des Abschnitts */
} EGPoint;
```

```
/* ATAK and RLSE chunks enthalten einen EGPoint[]-Vektor. In
   diesem sind die Dauer der jeweiligen Hüllkurvenabschnitte in ms
   sowie der Endlautstärkefaktor des Abschnitts als Festkommawert
   angegeben. */
```

Um eine gegebene Hüllkurve auf einen Sound aufzumodulieren, in diesem Fall eine Liste aus Lautstärke- und Zeitwerten, muß man sich folgender Programmiertechniken bedienen: Da nicht einfach die Lautstärkewerte der Liste nacheinander ausgegeben werden können, sondern fließende Übergänge zwischen ihnen geschaffen werden müssen, ist es unpraktisch, dies von einem normalen Programm erledigen zu lassen, welches ständig die Gesamtlautstärke berechnet und verändert. Vielmehr sollte man in einem bestimmten Zeitabstand zwischen den Werten der Liste interpolieren (etwa mit Hilfe der Geradengleichung) und die Gesamtlautstärke korrigieren, was z.B. in einem Interrupt geschehen kann. Im allgemeinen genügt es, wenn dies ca. alle 10 ms ausgeführt wird.

Die Abarbeitung der Hüllkurve beginnt mit dem Anschlag der Taste bzw. der Note im ATAK-Chunk bei der Lautstärke 0. Dann wird so lange zwischen den Tabellenwerten interpoliert, bis der letzte Wert erreicht ist. Dieser wird dann so lange gehalten, bis die Taste wieder losgelassen wird bzw. die Note endet. Dann beginnt die Abarbeitung der Hüllkurvendaten im RLSE-Chunk. Diese werden bearbeitet, bis das Ende der Liste erreicht ist (sollte mit Lautstärke 0 enden) oder mit derselben Stimme bzw. Kanal eine neue Note gespielt wird. Bei einem Playerprogramm ist es auch möglich, mit der Abarbeitung der RLSE-Hüllkurve schon vor dem eigentlichen Ende der Note zu beginnen, damit dieses mit dem Erreichen des Tabellenendes zusammenfällt. Dann sollte die ATAK-Hüllkurve rechtzeitig abgebrochen werden, falls die Note kürzer ist als die Dauer der ATAK- und RLSE-Hüllkurventeile zusammen.

Abb 3.2 zeigt ein Beispiel für eine aus einem ATAK- und einem RLSE-Chunk zusammengesetzte Hüllkurve. Beide enthalten je vier *EGPoint*-Einträge, was sich aus *ckSize/sizeof(EGPoint)* errechnet. Der Lautstärkewert in der Sustain-Phase entspricht dem letzten Eintrag in der ATAK-Liste, bis die Abarbeitung der RLSE-Liste erfolgt.

Der BODY-Chunk

Der BODY-Chunk enthält die Daten der Wellenformen aller Oktaven, die für diese 8SVX FORM defniniert sind.

```
#define ID_BODY      MakeID('B', 'O', 'D', 'Y')
/* enthält einen BYTE[]-Vektor aus 8-Bit-Audiodaten */
```

Innerhalb des BODY-Chunks sind die Audiodaten im Zweierkomplement nach Oktaven geordnet abgelegt, wobei die höchste und kleinste Oktave am Anfang steht. Die Daten der folgenden Oktave sind jeweils doppelt solang wie die der vorhergehenden. Eine einzelne Oktave besteht aus einem OneShot- und einem Repeatpart, oder ggf. nur einem von beiden. Abb 3.3 zeigt ein Beispiel für diese Anordnung. Hier könnte z.B. *oneShotHiSamples* = 24, *repeatHiSamples* = 16, *samplesPerHiCycle* = 8 und *ctOctave* = 3 gesetzt sein. Die durchgehenden Balken trennen Oktaven, die halben trennen jeweils OneShot- und Repeat-Part und die kleinen Balken repräsentieren die einzelnen Zyklen.

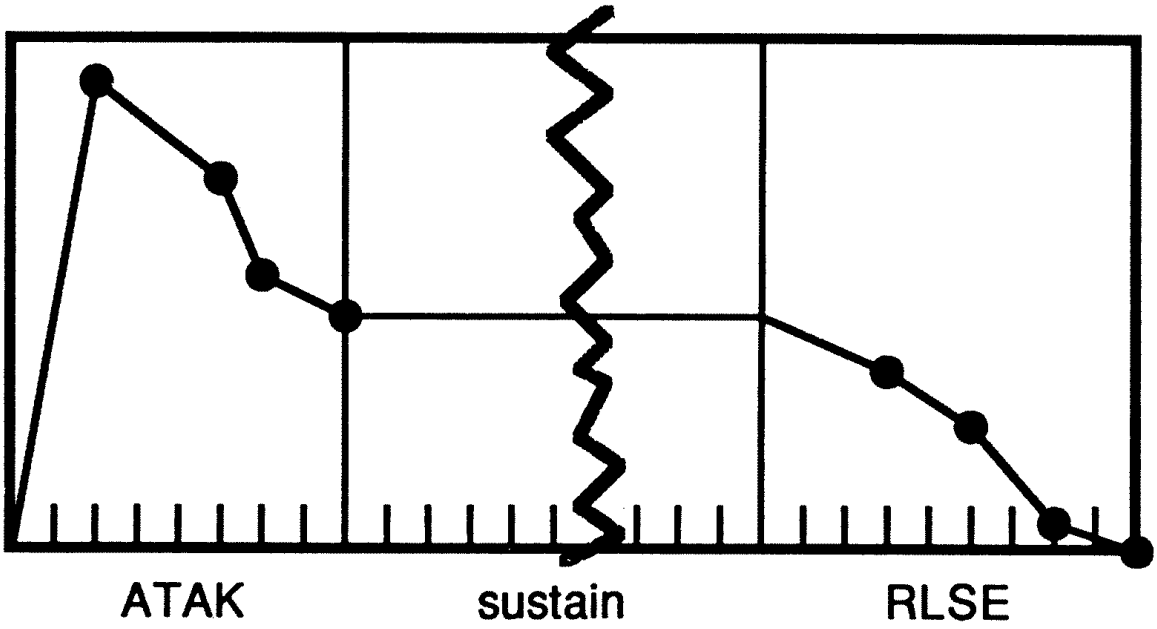


Abb. 3.2: ATAK- und RLSE-Hüllkurve

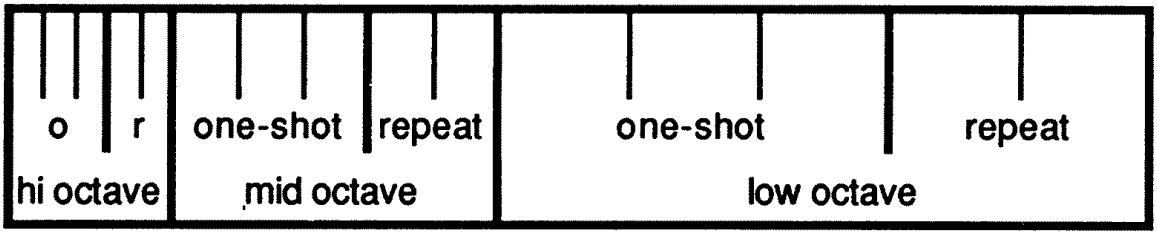


Figure 2. BODY subdivisions.

Abb. 3.3: Aufteilung der Daten im BODY-Chunk

Im allgemeinen enthält ein BODY-Chunk so viele Oktaven, wie in *ctOctave* angegeben ist. Die Daten für die höchste Oktave stehen am Beginn des Chunks, und enthalten am wenigsten Samples: *oneShotHiSamples+repeatHiSamples*. Jede nachfolgende Oktave enthält zwar die gleiche Anzahl Zyklen, ist aber jeweils doppelt so lang. Die Länge der letzten Oktave im Chunk ist dann:

$$(2^{0} + \ldots + 2^{\mathbf{ctOctave-1}}) \cdot (\mathbf{oneShotHiSamples} + \mathbf{repeatHiSamples})$$

Formel 51

Um Spannungssprünge und »Clicks« zu vermeiden, sollte das Ende des OneShot-Parts fließend in den Beginn des Repeatparts übergehen, d.h.; es sollten keine größen Wertunterschiede an dieser Stelle auftreten. Auch sollte das Ende des Repeat -Parts wieder weich in dessen Beginn übergehen.

Derartige Klangdaten mit OneShot- und Repeat-Teil können mit Hilfe der Amiga-Audiohardware sehr einfach wiedergegeben werden: Zuerst wird die Adresse des OneShot-Parts als Datenpointer und und (*oneShotHiSamples+repeatHiSamples*) als Länge der Daten an die Register übergeben und dann die DMA gestartet. Danach braucht nur noch die Adresse des Repeat-Parts und *repeatHiSamples* als Länge übergeben zu werden, damit ab der nächsten Wiederholung nur noch dieser gespielt wird.

Wenn ein Kompressionsalgorithmus gewählt ist, also *sCompression* im VHDR-Chunk ungleich *sCmpNone* ist, muß das Datenfeld des BODY-Chunk zuerst als ganzes dekomprimiert werden. Alle Angaben im VHDR-Chunk, wie die Länge von OneShot- und Repeat-Teil beziehen sich auf den dekomprimierten Zustand. In jedem Fall sollte ein BODY-Chunk mit ungerader Länge mit einem Pad-Byte (0) abgeschlossen werden.

Fibonacci-Delta-Codierung

Dieser Algorithmus wurde von Steve Hayes für die Kompression der beim Amiga üblichen 8-Bit-Samples entwickelt. Hier werden wie bei der normalen Delta-Codierung die Differenzen zwischen aufeinanderfolgenden Werten gebildet und gespeichert. Um auch hohe Anstiegsraten erzielen zu können, wird das Ergebnis jedoch erst in einen Index auf eine Tabelle umgewandelt und damit auf 4 Bit verkürzt. Dies reduziert den Speicherbedarf einer so komprimierten Wellenform zwar auf die Hälfte (plus 2 Byte), es werden jedoch nur kleine Differenzen korrekt wiedergegeben, bei höheren ergeben sich durch die geringere Werteauflösung Verzerrungen. Deshalb sollte diese Technik mit Vorsicht angewandt und in jedem Fall zuerst ausprobiert werden.

Hier das C-Listing für den Dekrompressionsalgorithmus: (Listing stammt aus [Addison-Wesley: Amiga – ROM Kernel Reference Manual / Includes&Autodocs S. I–72])

```
/* DUnpack.c --- Fibonacci Delta decompression by Steve Hayes */

#include <exec/types.h>

/* Fibonacci delta encoding for sound data */
BYTE codeToDelta[16] = {-34,-21,-13,-8,-5,-3,-2,-1,0,1,2,3,5,8,13,21};

/* Dekomprimiert Fibonacci-Delta-kodierte Daten aus einem n Byte
   langen Quellbereich in einen 2*n Byte langen Zielbereich, unter
   Angabe eines Anfangswertes x. Gibt den letzten Datenwert x zurück,
   sodaß die Funktion mehrfach aufgerufen und so ein Speicherbereich
   Stück für Stück dekomprimiert werden kann.
 */

BYTE D1Unpack(source,n,dest,x)
BYTE source[], dest[];
LONG n;
BYTE x;
   {
```

```
   BYTE d;
   LONG i, lim;

   lim = n << 1;
   for (i=0; i < lim; ++i)
      {
      /* erst high nibble, dann low nibble decodieren*/
      d = source[i >> 1];     /* zwei (ein  Byte) holen */
      if (i & 1)              /* high oder low nibble */
         d &= 0xf;            /* low: ausmaskieren */
      else
         d >>= 4;             /* high: um 4 rechtsschieben */
      x += codeToDelta[d];    /* Tabellenwert aufaddieren*/
      dest[i] = x;            /* neuen Wert speichern */
      }
   return(x);
   }

/* Dekomprimiert Fibonacci-Delta-codierte Daten aus einem n Byte
   langen Quellbereich in einen 2*n Byte langen Zielbereich. Der
   Quellpuffer besteht aus einem Pad-Byte, einem 8-Bit Anfangswert,
   gefolgt von n-2-Bytes, die 2*(n-2) 4-Bit-Samples enthalten.
 */

void DUnpack(source, n, dest)
BYTE source[], dest[];
LONG n;
   {
   D1Unpack(source+2, n-2, dest, source[1]);
   }
```

3.1.2 FORM-8SVX-Schnellübersicht

Typ-Definitionen

Hier eine Zusammenfassung sämtlicher C-Definitionen für die 8SVX-FORM. Strukturelemente werden in der beschriebenen Reihenfolge aneinandergereiht. Ein "UBYTE"-Feld wird zu einem 8-Bit Byte zusammengefasst. Programme sollten alle Pad-Elemente auf 0 setzen.

```
#define ID_8SVX       MakeID('8', 'S', 'V', 'X')
#define ID_VHDR       MakeID('V', 'H', 'D', 'R')

/* ---------- Voice8Header ----------------------------------------------*/
typedef LONG Fixed;          /* Festkommawert mit 16 Bit links und 16 Bit
                                rechts vom Komma. Kann gesehen werden als
                                eine Zahl aus 2^16teln, also 65536teln. */
#define Unity 0x10000L       /* Unity = Fixed 1.0 = maximale Lautstärke */

/* sCompression: gewählter Kompressions-Algorithmus */
#define sCmpNone        0    /* nicht komprimiert */
#define sCmpFibDelta    1    /* Fibonacci-delta-Kodierung */
                             /* Weitere Kompressionsarten möglich */
```

Taken from Amiga-Manuals-Website

```
typedef struct {
   ULONG oneShotHiSamples,  /* # Samples im OneShotPart (höchste Okt.)*/
         repeatHiSamples,   /* # Samples im RepeatPart (höchste Okt.)*/
         samplesPerHiCycle; /* # Samples/Cycle (höchste Okt),sonst 0 */
   UWORD samplesPerSec;     /* Sampling-Rate */
   UBYTE ctOctave,          /* Anzahl der Oktaven im BODY-Chunk */
         sCompression;      /* gewählter Kompressions-Algorithmus */
   Fixed volume;            /* Lautstärke von 0 bis Unity (1.0, volle Lautst.) */
} Voice8Header;


#define ID_Name MakeID('N','A','M','E')
/* enthält einen char[]-Vektor (den Namen des Instruments) */

#define ID_CopyRight MakeID('(','c',')',' ')
/* enthält einen char[]-Vektor (den Copyright-Hinweis) */

#define ID_AUTH MakeID('A','U','T','H')
/* enthält einen char[]-Vektor (den Namen des Autors) */

#define ID_ANNO MakeID('A','N','N','O')
/* enthält einen char[]-Vektor (Erstellungsdatum o.ä.) */



#define ID_ATAK      MakeID('A', 'T', 'A', 'K')
#define ID_RLSE      MakeID('R', 'L', 'S', 'E')

/* ---------- Envelope ATAK & RLSE ------------------------------------*/
typedef struct {
   UWORD duration;     /* Dauer d. Hüllkurvenabschnitts in ms, > 0 */
   Fixed dest;         /* Lautstärkefaktor am Ende des Abschnitts */
} EGPoint;

/* ATAK and RLSE chunks enthalten einen EGPoint[]-Vektor. In
   diesem sind die Dauer der jeweiligen Hüllkurvenabschnitte in ms
   sowie der Endlautstärkefaktor des Abschnitts als Festkommawert
   angegeben. */


#define ID_BODY      MakeID('B', 'O', 'D', 'Y')
/* enthält einen BYTE[]-Vektor aus 8-Bit-Audiodaten */
```

8SVX-Standardausdrücke

Hier eine Liste der Standardausdrucke der 8SVX FORM Syntax. Eine 8SVX FORM kann ein selbständiges IFF-File oder ein Teil eines solchen sein.

```
8SVX       ::= "FORM" #{ "8SVX" VHDR [NAME] [Copyright] [AUTH] ANNO*[ATAK] [RLSE] BODY
}

VHDR       ::= "VHDR" #{ Voice8Header  }
NAME       ::= "NAME" #{ CHAR*         } [0]
```

Taken from Amiga-Manuals-Website

```
Copyright  ::= "(c) " #{ CHAR*        } [0]
AUTH       ::= "AUTH" #{ CHAR*        } [0]
ANNO       ::=."ANNO" #{ CHAR*        } [0]

ATAK       ::= "ATAK" #{ EGPoint*     }
RLSE       ::= "RLSE" #{ EGPoint*     }
BODY       ::= "BODY" #{ BYTE*        } [0]
```

Das Zeichen "#" steht für das *ckSize*-Feld (LONG, 32 Bits!), also die Länge der Daten innerhalb der geschweiften Klammern { }. Buchstaben wie z.B. ID werden in Anführungszeichen " " gesetzt, was in eckigen Klammern [ ] steht ist optional, und "*" bedeutet 0, eine oder mehrere Wiederholungen des Ausdrucks davor. Ein manchmal benötigtes Pad-Byte wird immer als [0] angegeben.

Die Reihenfolge der Chunks muß im 8SVX-Format nicht so streng an der obigen Standardschreibweise ausgerichtet sein. Die Chunks VHDR, Name, Copyright und AUTH können in beliebiger Reihenfolge auftreten, so lange sie noch vor dem BODY-Chunk stehen. Die Chunks ANNO, ATAK und RLSE müssen nicht unbedingt vor dem BODY-Chunk stehen. Natürlich ist in Zukunft mit noch mehr Chunk-Typen der 8SVX-FORM zu rechnen.

Taken from Amiga-Manuals Website

Die SMUSPlayer-Library

4.1 Einführung

Der Amiga bietet mit seinen hervorragenden Soundfähigkeiten einen großen Anreiz, eigene Programme mit möglichst ausgefeilten Musik- und Geräuscheffekten zu untermalen. Dies zu erreichen war jedoch bisher sowohl mit großem Programmieraufwand als auch mit tieferer Kenntnis der Hardware verbunden. Eine einfachere Möglichkeit, komplexe Musikeffekte zu erzielen, bieten die gängigen Kompositionsprogramme, wie z.B. »Deluxe Music Construction Set« von ECA, »Sonix« von Aegis Software , etc. Diese jedoch erlauben meist nur das Abspielen ihrer Kompositionen innerhalb des Programms selbst, was deren Verwendung innerhalb eigener Programme unmöglich macht. Eine einfache und bequeme Art, Ihre musikalischen Werke unabhängig von den jeweiligen Editorprogrammen erklingen zu lassen, bietet das Programmpaket SMUSPlayer. Der Kern von SMUSPlayer ist eine Bibliothek von Unterroutinen, im Englischen »Library« genannt, welche Ihnen die Einbindung Ihrer Eigenkompositionen auf denkbar einfache Weise gestattet. Im folgenden Abschnitt soll zuerst das Prinzip der Library, welches einen wesentlichen Bestandteil der gesamten Konzeption des Amiga-Betriebssystems darstellt, näher erläutert werden.

4.1.1 Was ist eine Library?

Eine Library besteht im wesentlichen aus einer Library-Struktur, einer Sprungtabelle und einem Programmteil. In der Struktur sind u.a. ein Node, welcher die Einbindung in eine verkettete Liste erlaubt, zwei Worte für die Größe von Struktur und Sprungliste, die Versionsnummer und ein Zähler für die Anzahl der öffnenden Tasks enthalten. Die Sprungtabelle besteht aus Einträgen von je 6 Byte, von denen zwei den MaschinenBefehl 'JMP' und vier die absolute Adresse der entsprechenden Routine im Programmteil enthalten. Die ersten vier Einträge direkt unterhalb der Struktur sind für interne Zwecke reserviert, darunter beginnen die für den User nutzbaren Programmeinsprünge.

Versucht man eine Library mit der Exec-Routine OpenLibrary zu öffnen, so sucht das System zuerst in der Resident-Liste nach dem angegebenen Namen. Diese Liste wird bei jedem Reset neu erstellt und enthält Namen und Position aller im ROM befindlichen Libraries, Devices, Resources etc. Wird dort nichts gefunden, so setzt das System die Suche im logischen Laufwerk »libs:« fort, welches normalerweise dem Verzeichnis »libs:« auf der Bootdiskette zugeordnet ist. Wird dort eine Datei mit dem gesuchten Namen gefunden, so wird diese wie ein Executable geladen, d.h., sie wird wie ein normales Programm nach Code-, Data- und BSS-Segmenten durchsucht und anschließend reloziert. Da sich solche Library-Dateien äußerlich (zumindest für das DOS) in

nichts von Programmdateien unterscheiden, besteht die erste als Programm ausgeführte Position aus einem Befehl, der einen Fehlercode zurückgibt und dann das »Programm« beendet, damit eine solche versehentliche Ausführung nicht zum Systemabsturz führt.

Ist die Datei nun glücklich im Speicher, wird die Sprungtabelle erstellt, die Struktur initialisiert und mit dem Node in die Library-Liste eingefügt. Die Routine *OpenLibrary()* kehrt dann mit einem Zeiger auf die Struktur zurück, über den man mit negativen Offsets direkt die Einträge der Jumptabelle anspringen kann. Wird die Library nun von weiteren Tasks geöffnet, so gibt OpenLibary() wie vorher den Library-Zeiger zurück und erhöht nur noch einen Zähler. Erst wenn dieser durch entsprechend viele Aufrufe von CloseLibrary() wieder auf 0 heruntergezählt wurde, wird die Library aus dem System entfernt und der reservierte Speicherplatz wieder freigegeben.

4.1.2 Die SMUSPlayer-Library

Libraries erlauben also die Auslagerung nahezu beliebig großer und komplexer Unterprogrammsammlungen auf Diskette. Wichtig dabei ist, daß sich hierbei gegenüber den ROM-residenten Libraries keinerlei Einschränkungen ergeben, vor allem was die Benutzung in verschiedenen Programmiersprachen betrifft. Da sich jede Implementation einer Sprache auf dem Amiga der ROM-Libraries bedienen können muß, allein schon um so grundlegende Aktivitäten wie Ein-/Ausgabe etc. ausführen zu können, steht auch der Nutzung aller anderen Libraries im allgemeinen nichts im Wege. Diese Gesichtspunkte vor allem waren es, die für die Konzeption des Players als Library den Ausschlag gaben.

Der SMUS-IFF-Standard

Die Zielsetzung des SMUSPlayers ist, eine Möglichkeit zu schaffen, die abgespeicherten Scores (Lieder) möglichst vieler gängiger Kompositionsprogramme von eigenen Programmen aus gesteuert wiederzugeben, unabhängig von der jeweiligen Programmumgebung. Die meisten dieser Programme bedienen sich bei Wiedergabe und Speichern eines Scores des SMUS-Formats. Der SMUS IFF-Standard (Simple MUSical Score Interchange File Format) ist ein Teil des IFF-Standards, welcher von der Firma Electronic Arts[tm] entwickelt und von Commodore Amiga allen Softwareentwicklern zur Vereinheitlichung ihrer Produkte vorgeschlagen wurde, um eine weitgehende Kompatibilität zwischen den verschiedenen Grafik-, Sound-, Animations- und Textverarbeitungsprogrammen herzustellen. Dies erwies sich zwar in einigen dieser Bereiche für schwer durchführbar, im Bereich der Klangverarbeitung jedoch scheint sich einstweilen das SMUS-Format für die Darstellung von musikalischen Daten weitgehend durchgesetzt zu haben.

Die Funktionen des SMUSPlayers

LoadScore und PlayScore

Die einfachste Art, mit dem SMUSPlayer Musik in eigene Programme zu integrieren, ist das einfache Laden und Abspielen der gewünschten Scores. Diesen Zweck erfüllen die Routinen *LoadScore* und *PlayScore*. Mit *LoadScore* stellen Sie der Library die Daten des wiederzugebenden Scores zur Verfügung, indem Sie im Prozessorregister A0 einen Zeiger auf den Namen des abzuspielenden Scores und in A1 einen Zeiger auf den Namen des Verzeichnisses übergeben, in

welchem die dem Score zugehörigen Instrumente enthalten sind. Die Namen müssen den normalen DOS-Konventionen entsprechen und mit einer 0 enden. Gelingt es, den Score einzulesen, so wird in D0 eine 0 zurückgegeben, anderfalls –1.

Um den von einem Score und seinen Instrumenten belegten Speicher wieder freizugeben, verwenden Sie die Routine *UnLoadScore*. Dies stoppt die Wiedergabe des Scores und der benötigte Speicher wird an das System zurückgegeben. *UnLoadScore* wird beim Schließen der Library automatisch aufgerufen.

Sollten Sie eine andere Programmiersprache als Assembler verwenden, so wird diese Übergabe von den jeweiligen Interfaceroutinen übernommen, Sie müssen dann nur noch den Übergabekonventionen der betreffenden Sprache Folge leisten.

Durch einen Aufruf von PlayScore veranlassen Sie das Abspielen der geladenen Daten. Hierbei übergeben Sie in D0 eine 0, um anzuzeigen, daß der mit *LoadScore* geladene Score gespielt werden soll. In D1 kann die Anzahl der Wiederholungen angegeben werden, also wie oft der Score hintereinander abgespielt werden soll. Eine 0 bewirkt die endlose Wiederholung des Scores.

Mit *Stop* können Sie die Wiedergabe anhalten. (Vorläufig: Der Aufruf von *Stop* während der Wiedergabe läßt die gespielten Töne ausschwingen, außerhalb der Wiedergabe bewirkt *Stop* sofortige Stille. Das bedeutet, daß zweimaliger Aufruf von *Stop* in jedem Fall sofort alle Stimmen abschaltet.)

LoadSet

Dieses Verfahren erlaubt das Abspielen, stoppen und beliebig oft Neustarten eines geladenen Scores. Möchte man jedoch mehrere verschiedene Geräuscheffekte oder auch längere Scores auf einmal zur Verfügung haben, so müßte bei jedem Wechsel des wiederzugebenden Liedes oder Klangeffekts ein Zugriff auf einen Datenträger erfolgen. Dies würde jedoch eine sofortige, reaktionsschnelle Wiedergabe (z.B. für untermalende Geräuscheffekte in einem Spiel) unmöglich machen. Zu diesem Zweck besitzt der SMUSPlayer eine Möglichkeit, ganze Sätze von abgespeicherten Scores zu laden und, unter Angabe einer dem einzelnen Effekt zugeordneten Nummer, abzuspielen. Diese Sätze werden mit dem Programm »SetPacker« erzeugt, welches sich auf der Systemdisk befindet. Der SetPacker kombiniert Ihre abgespeicherten Scores, prüft auf mehrfaches Vorkommen von Instrumenten, und speichert alles in einer einzelnen Datei ab. Durch dieses Verfahren verkürzen sich die Ladezeiten von Diskette auf angenehme Weise.

Diese gespeicherten Sätze (sie tragen das Suffix ».sset«) können nun mit Hilfe der Routine *LoadSet* dem System zur Verfügung gestellt werden, indem Sie in A0 einen Zeiger auf den Namen der Datei übergeben.

Für die Wiedergabe der Scores des geladenen Satzes ist ebenfalls die Routine *PlayScore* zuständig. An *PlayScore* übergeben Sie in D0 die Positionsnummer des Scores in der alphabetischen Ordnung im Satz und in D1 die Anzahl der Wiederholungen. Auch hier hält man mit *Stop* die Wiedergabe an.

Benötigt Ihr Programm in gewissen Phasen das *Set* nicht, so kann der durch den Satz belegte Speicher durch die Funktion *UnloadSet* wieder freigegeben werden. Selbstverständlich wird das ebenfalls beim Schließen der Library mit *CloseLibrary()* automatisch erledigt.

LoadAux

Möchte man ein externes, zusätzliches Instrument zur Verfügung stellen, sei es nun, weil spezielle Klangeffekte zu komplex sind, um nebeneinander im Speicher Platz zu finden, oder weil es einfach zu viele sind, so kann dies mit Hilfe der Routine *LoadAux* geschehen. Hierdurch kann ein IFF- oder RFF-(Sonix-)Instrument geladen und mit Hilfe der weiter unten beschriebenen Routinen verwendet werden. An LoadAux übergibt man in A0 einen Zeiger auf den File-Namen des Instruments. Ruft man *LoadAux* ein zweitesmal auf, so wird das zuvor geladene Instrument entfernt und durch das neue ersetzt. Deswegen kann sich vorläufig nur ein Aux-Instrument gleichzeitig im Speicher befinden.

Der vom Instrument belegte Speicher wird beim Schließen der SMUSPlayer.library wieder freigegeben.

FindInstr

Alle folgenden Routinen, denen in irgendeiner Form ein Instrument angegeben werden muß, suchen dieses mit Hilfe der Routine *FindInstr*. An *FindInstr* wird ein Zeiger auf den Namen des Instruments in A0 übergeben, und zwar diesmal nicht der File-Name, sondern der eigentliche Name des Instruments, der bei IFF-Files im NAME-Chunk der 8SVX-FORM und bei RFF-Instrumenten ab Adresse hex $24 im ».instr«-File steht (siehe Kapitel 3.2). Dieser sollte zwar stets mit dem File-Namen identisch sein, enthält aber keine Extensions wie eben z.B. ».instr«.

Wird *FindInstr* aufgerufen, so wird zuerst in einem evtl. mit *LoadScore* geladenen Score nachgesehen, ob das Instrument mit diesem in den Speicher gelangt ist. Wurde *LoadScore* nicht verwendet oder ist das Instrument dort nicht zu finden, so werden die Instrumente eines evtl. geladenen Sets durchsucht. Konnte auch hier kein Instrument dieses Namens gefunden werden, so gibt die Routine eine 0 in D0 zurück, im Erfolgsfall jedoch einen Zeiger auf das Instrument. Dieses kann dann, abhängig vom Instrumententyp, sogar noch vom eigenen Programm aus manipuliert werden. So könnten z.B. Sonix-Analoginstrumente noch mit zusätzlichen Effekten wie der dynamischen Veränderung von LFO-Frequenz, Portamento etc. versehen werden.

Wird *FindInstr* mit einer 0 in A0 aufgerufen, so wird ein Zeiger auf ein evtl. mit LoadAux geladenes Instrument zurückgegeben, sonst 0.

SetDefault

In einem SMUS-File sollte im Normalfall jeder Track (Stimme) mit einer Instrumentenzuordnung versehen sein, die den Klang der abgespielten Noten eindeutig definiert. Fehlt eine solche Zuordnung, so müßte der Track bei der Wiedergabe eigentlich stumm bleiben. Die meisten Kompositionsprogramme sehen jedoch eine automatische Instrumentenzuordnung für solche Fälle vor, das sog. Default-Instrument. Dieses kann in den meisten Fällen frei gewählt werden. Auch der SMUSPlayer besitzt eine Möglichkeit, um undefinierten Tracks ein Instrument zuzordnen. Dies geschieht mit Hilfe der Routine *SetDefault*. In A0 muß ein Zeiger auf den Namen des Instruments oder 0 für das Aux-Instrument angegeben werden, analog den Konventionen für die Routine *FindInstr* (diese wird auch von *SetDefault* zuerst aufgerufen). Von nun an werden alle Tracks ohne definiertes Instrument über dieses Instrument gespielt.

PlayNote und PlayTone

Die Möglichkeiten von SMUSPlayer sind aber nicht auf die bloße Wiedergabe von Scores beschränkt. Speziell für Fälle, in denen vom Score unabhängige Soundeffekte benötigt werden, z.B. für Animationen und Spiele, sind zwei Routinen zum Abspielen von einzelnen Instrumenten vorgesehen. Die eine, PlayNote, dient zur Wiedergabe einer Note, d.h. ein Instrument wird eine bestimmte Zeit lang mit einer bestimmten Tonhöhe gespielt. Die andere, PlayTone, ignoriert die Angabe der Notenlänge und spielt das Instrument so lange, bis es durch die Routine ReleaseTone wieder stummgeschaltet wird.

Um diese Routinen mit den benötigten Parametern zu versorgen, muß man in A0 einen Zeiger auf eine *SPNote*-Struktur übergeben. In ihr ist u.A. eine *SNote*-Struktur enthalten, wie sie in den *TRAK*-Chunks des SMUS-Formats üblich ist. Die C-Definition der *SPNote*-Stuktur lautet:

```
struct SPNote {
    char *iName;
    struct SNote SNote;
    UWORD vol;
    UBYTE voice;
    UBYTE pri;
};
```

Die Elemente dieser Struktur haben die folgenden Bedeutungen:

iName	Zeiger auf den Namen des Instruments.
SNote	SNote – Struktur
vol	Lautstärke
voice	Kanalnummer (1 – 4)
pri	Priorität der Note (–128 bis 127)

Der Namensstring, auf den *iName* zeigt, wird an *FindInstr* übergeben (siehe dort). Es können also die durch *LoadScore* oder *LoadSet* zur Verfügung gestellten Instrumente verwendet werden, oder auch das Aux-Instrument.

In der *SNote*-Struktur sind die Tonhöhe (MIDI-Pitch-Nummer) und die Informationen über den Notenwert enthalten (siehe 2.2.4.1). Die effektive Spieldauer einer Note richtet sich u.A. auch nach dem Tempo des gerade bzw. zuletzt gespielten Scores. *PlayTone* ignoriert den Notenwert. Die Felder *chord* und *tieout* haben hier keine Bedeutung.

Das *voice*-Feld enthält die Nummer des Kanals, auf dem das Instrument gespielt werden soll. Diese entspricht der normalen Numerierung der Kanäle plus eins, also von 1 bis 4. Dies wurde so gewählt, da eine 0 in diesem Feld die freie Wahl eines Kanals durch den SMUSPlayer bewirken soll. Wenn im *voice*-Feld ein Wert von 1 bis 4 angegeben ist, so wird versucht, die Note auf dem entsprechenden Kanal zu spielen. Ist der Kanal frei, so gelingt dies auch und der Kanal wird für die Dauer der Note mit der in *pri* angegeben Priorität belegt. Nach dem Aufruf dann steht in D0 die Nummer des Kanals (ebenfalls von 1 bis 4, wie in voice). Wird auf diesem Kanal jedoch gerade eine andere Note mit gleicher oder höherer Priorität gespielt, so kehrt die Routine mit –1 als Fehlermeldung in D0 zurück.

Wenn das *voice*-Feld jedoch eine 0 enthält, sucht sich der SMUSPlayer selbst einen passenden Kanal. Hierzu geht er folgendermaßen vor: Zuerst wird nach einer ganz freien Stimme gesucht, die weder von einem Score noch von einem anderen Aufruf von *PlayNote* oder *PlayTone* besetzt sind. Ist ein solcher nicht verfügbar, so durchsucht er die momentanen Prioritäten auf den Kanälen nach einem Wert, der niedriger ist als der eigene. Erst wenn auch in diesem Durchgang nichts gefunden wurde, wird die Suche abgebrochen und –1 in D0 zurückgegeben. Ansonsten wird die Nummer des gefundenen Kanals in D0 zurückgegeben und die Note auf diesem gespielt.

Im *pri*-Element muß die Priorität angegeben werden, mit der der jeweilige Kanal während der Wiedergabe der Note belegt wird. *PlayNote* und *PlayTone* können bereits besetzte Kanäle nur dann übernehmen, wenn im *pri*-Feld eine höhere Priorität steht als die am Kanal anliegende. Gleiche Prioritäten unterbrechen sich nicht untereinander.

Hinweis: Alle durch die Score-Wiedergabe gespielten Noten haben die Priorität 0. Möchte man eine hierdurch belegte Stimme nutzen, so muß man im *pri*-Feld eine Priorität von mindestens 1 angeben.

SetReturnSig

Die Verwendung dieser Routine ermöglicht die Synchronisation eines Programmes mit dem Ende eines Scores. Erreicht die Wiedergabe eines mit PlayScore gestarteten Scores dessen Ende, so kann damit das Absenden eines Signals an einen oder mehrere Tasks bewirkt werden. Dieses Signal muß der jeweilige Task zuvor angeben, in dem er *SetReturnSig* mit der Signalmaske in D0 aufruft. In dieser Signalmaske können auch mehrere Bits gesetzt sein. Der Task kann nun auf das Eintreffen dieses Signals warten, oder auch einen Softwareinterrupt auf dieses Ereignis hin bestimmen. Gibt man eine 0 in D0 an und ruft *SetReturnSig* auf, löscht die Library alle Bits der Signalmaske für den jeweiligen Task und entfernt diesen aus der Liste der Signalempfänger.

Die Kompositionsprogramme

Im Prinzip kann SMUSPlayer die abgespeicherten Scores eines jeden Kompositionsprogrammes wiedergeben, welches sich an den SMUS IFF-Standard hält. Es gibt zu diesem Standard allerdings bei einigen Programmen gewisse Erweiterungen, welche die Wiedergabe ihrer Scores durch andere Programme verhindern. Ein solches ist z.B. Sonix[tm] von Aegis Software. Sonix kann zwar die Dateien der meisten Kompositionsprogramme lesen, umgekehrt gilt das leider für die meisten Programme nicht. Daher wurde bei der Entwicklung von SMUSPlayer der vollständigen Kompatibilität zu Sonix größte Aufmerksamkeit geschenkt.

Es ist nicht die Zielsetzung von SMUSPlayer, ein eigenes Kompositions- und Editorsystem zu bilden, wie es für den Entwurf eigener Musikstücke notwendig ist.

4.2 Die Programmierung der Library

Wenn Sie es noch nicht getan haben sollten, so schlagen Sie bitte zuerst das Kapitel Installation auf, und folgen den Anweisungen des entsprechenden Abschnitts, um Ihre Arbeitsumgebung entsprechend zu konfigurieren.

4.2.1 Die Programmierung der Library in Assembler

Bevor Sie auf die Library zugreifen können, muß diese erst mit *OpenLibrary()* geöffnet und dann der Zeiger auf ihre Struktur zwischengespeichert werden. Dies kann beispielsweise durch folgende Sequenz geschehen:

```
        INCLUDE "SMUSPlayer_lib.i"
        INCLUDE "exec/exec_lib.i"

        lea         LibName,a1
        clr.l       d0
        move.l      a6,-(a7)
        movea.l     4,a6
        jsr         LVOOpenLibrary(a6)
        move.l      (a7)+,a6
        move.l      d0,SMUSPlayerBase
        ...

SMUSPlayerBase      ds.l    1
LibName             ds.b    "smusplayer.library",0
```

Sie können allerdings auch die komfortablen Makros aus der Include-Datei *asmsupp.i* verwenden, was in diesem Fall so aussähe:

```
        lea         LibName,a1
        clr.l       d0
        CALLSYS     OpenLibrary
        move.l      d0,SMUSPlayerBase
```

Nun ist die Library initialisiert und steht Ihnen zur Verfügung. Zum Aufruf einer Routine übertragen Sie zuerst die benötigten Parameter in die jeweiligen Register und dann den Library-Pointer in A6, um über diesen in die gewüschte Routine einszupringen. Um beispielsweise einen Score zu laden, der auf der Systemdiskette von Sonix abgespeichert ist, verwenden Sie die Sequenz:

```
        ...
        lea         ScoreName,a0
        lea         InstDirName,a1
        move.l      a6,-(a7)
        movea.l     SMUSPlayerBase,a6
        jsr         _LVOLoadScore(a6)
        move.l      (a7)+,a6
        tst.l       d0
        bne         Error
        ...
```

Auch hierfür ist in asmsupp.i ein Macro vorgesehen, nämlich LINKSYS:

```
        lea         ScoreName,a0
        lea         InstDirName,a1
        LINKSYS     LoadScore,SMUSPlayerBase
```

Taken from Amiga-Manuals-Website

Genauso brauchen Sie jetzt z.B. für einen Aufruf von *Stop* nur noch die Zeile

```
LINKSYS     Stop,SMUSPlayerBase
```

einzugeben, da *Stop* weder Parameter benötigt noch irgendwelche Ergebnisse zurückgibt.

Um die Library wieder zu schließen (was Sie als ordentlicher Programmierer stets tun sollten), verwenden Sie die Zeilen:

```
lea         SMUSPlayerbase,a1
CALLSYS     CloseLibrary
```

4.2.2 Die Programmierung der Library in C

Die Interfaceroutinen von C benötigen, um auf die Library zugreifen zu können, ein globales Symbol, genannt Librarybase. Dieses Symbol muß ein Zeiger auf die im entsprechenden Header-File definierte Library-Struktur sein, welche stets eine Erweiterung der Standard-Library-Strukur bildet. Der Zeiger muß von Ihrem Programm zur Verfügung gestellt und mit dem Ergebniswert von *OpenLibrary()* initialisiert werden. Dies könnte z.B. durch folgenden Programmkopf realisiert werden:

```
#include <smusplayer_lib.h>

struct SMUSPlayerBase *SMUSPlayerBase;

main()
{
 if(!(SMUSPlayerBase = OpenLibrary("smusplayer.library",0)))       {
    printf("SMUSPlayer.lib konnte nicht geöffnet werden!\n");
}
...
```

Wenn Sie alle diese Elemente in Ihrem Programm untergebracht haben, können die Funktionen, wie bei allen anderen Libraries auch, durch einfache Angabe des Namens und der Parameter in Klammern aufgerufen werden, z.B.

```
char *ScoreName;
char *InstDirName;
BOOL Error;

Error = LoadScore(ScoreName,InstDirName);
if(Error) {
   printf("Score %s konnte nicht geladen werden!\n");
}
```

Wie bei allen Programmiersprachen, so sollten Sie auch hier die Library ordnungsgemäß schließen, z.B. mit der Zeile:

```
CloseLibrary(SMUSPlayerBase);
```

Von jetzt an müssen Sie nur noch darauf achten, daß Sie stets die Datei mit den Interfaceroutinen, *SMUSPlayer_lib.o*, im Linkerkommando berücksichtigen. Entweder schließen Sie sie als nor-

male Objektdatei mit ein (d.h. die gesamte Datei wird mit in Ihr Programm gelinkt), oder Sie wandeln diese mit den vom Compiler gestellten Tools in eine »Linked Library« um. Sie können diese dann mit der »-l«-Option des Linkers einbinden, wobei dann nur die wirklich benötigten Teile extrahiert werden.

4.2.3 Die Programmierung der Library in Basic

Die Verwendung der Library von Basic aus ist denkbar einfach. Nachdem die Datei *SMUSPlayer.bmap* an ihren Platz gebracht worden ist (siehe Kapitel Installation), brauchen Sie nur noch die benötigten Funktionen mit der *DECLARE FUNCTION*-Anweisung zu deklarieren und die Library mit der *LIBRARY*-Anweisung zu öffnen. Dies geschieht auf folgende Weise:

```
DECLARE FUNCTION LoadScore& LIBRARY
DECLARE FUNCTION UnLoadScore& LIBRARY
DECLARE FUNCTION LoadSet& LIBRARY
DECLARE FUNCTION UnloadSet& LIBRARY
DECLARE FUNCTION PlayScore& LIBRARY
DECLARE FUNCTION Stop& LIBRARY
DECLARE FUNCTION LoadAux& LIBRARY
DECLARE FUNCTION SetDefault& LIBRARY
DECLARE FUNCTION PlayNote& LIBRARY
DECLARE FUNCTION PlayTone& LIBRARY
DECLARE FUNCTION ReleaseTone& LIBRARY
DECLARE FUNCTION FindInstr& LIBRARY
DECLARE FUNCTION SetReturnSig& LIBRARY

LIBRARY "smusplayer.library"
```

Hierbei wurden der Einfachheit halber sämtliche Funktionen deklariert. Sie brauchen natürlich nur diejenigen aufzuführen, die Ihr Programm auch tatsächlich verwendet.

Der Aufruf erfolgt am besten mit Hilfe der *Call*-Anweisung mit dem Namen der Funktion dahinter, wobei das Schlüsselwort *Call* eine Verwechslung des Funktionsnamens mit einer Sprungmarke verhindert.

4.3 Das Programm SetPacker

Wie im Einführungsabschnitt bereits erläutert, ist es eine der wesentlichen Eigenschaften des SMUSPlayers, mehrere Scores parallel im Speicher zu halten und diese bei Bedarf wiedergeben zu können. Hierzu müssen Sie zu Sätzen, im folgenden »Sets« genannt, zusammengefaßt werden. Diese Aufgabe wird durch das Programm *SetPacker* erledigt, welches sich im Verzeichnis »SDK« Ihrer Systemdiskette befindet. Der *SetPacker* ermöglicht Ihnen die bequeme Auswahl der Musik- und Geräuscheffekte, die Sie in Ihrem Programm benötigen.

4.3.1 Der Start von SetPacker

Sie können *SetPacker* sowohl vom CLI als auch von der Workbench aus starten. Wenn Sie bevorzugt mit der Workbench arbeiten, gehen Sie bitte so vor:

Falls das noch nicht geschehen ist, laden Sie zuerst die Workbench mit *loadwb* und sorgen dafür, daß sich die SMUSPlayer-Systemdisk in einem der Laufwerke befindet. Öffnen Sie das Root-Fenster der Diskette durch einen Doppelklick auf ihr Icon.

Betätigen Sie nun das Icon des Verzeichnisses SDK mit einem Doppelklick. Im jetzt geöffneten Fenster sehen Sie das Icon des Setpackers, welches Sie durch einen weiteren Doppelklick starten können. Beachten Sie bitte, daß bei einem Start von der Workbench keine Fehler- oder sonstige Programm-Meldungen erscheinen; diese werden nur bei einem Start vom CLI in das dazugehörige Fenster ausgegeben.

Start vom CLI aus:

Begeben Sie sich mit dem Befehl »cd AEP:SDK« in das Root-Directory der SMUSPlayer-Systemdisk und starten Sie den SetPacker, indem Sie einfach »SetPacker« eingeben.

4.3.2 Das Auswahlfenster

Nach dem Start öffnet SetPacker ein Fenster, welches einen Bereich für die Darstellung und Anwahl der Dateinamen (Filebox) sowie verschiedene Gadgets beinhaltet. Die Bedeutung und Funktion dieser Elemente soll im folgenden näher erläutert werden:

Die Filebox

Die Filebox stellt das Haupteingabemedium für die Auswahl der einzelnen Scores dar. In ihr werden die Namen der Dateien, die im derzeit angewählten Verzeichnis enthalten sind, dargestellt. Es werden jedoch nur diejenigen Dateien angezeigt, die durch die Endung .smus als SMUS-Score-Dateien erkannt werden.

Haben Sie mit Hilfe der Gadgets ein Verzeichnis gewählt, so wird dessen Inhalt nach dem Einlesen zeilenweise aufgelistet. Sind mehr Einträge enthalten als gleichzeitig dargestellt werden können, so können Sie diese mit Hilfe der Schiebeleiste am Rand der Filebox sichtbar machen. Klicken Sie hierzu auf den Schiebebalken und ziehen diesen mit gedrückter Auswahltaste in die gewünschte Richtung, um den Darstellungsbereich entsprechend zu verschieben.

Es gibt zwei Gruppen von Einträgen: Score-Dateien und Verzeichnisse. Beide Gruppen erscheinen gemischt in alphabetischer Reihenfolge, unterscheiden sich jedoch farblich voneinander. Sie können das aktuelle Verzeichnis wechseln, indem Sie die Maus über den gewünschten Eintrag bewegen und diesen mit einem Doppelklick anwählen.

Um nun einen Score-Eintrag für die Übertragung in Ihr Set auszuwählen, klicken Sie ihn einmal mit der Maus an. Sie werden bemerken, daß dieser jetzt in inverser Schrift dargestellt ist. Gleichzeitig erscheint links neben dem Eintrag eine Nummer. Diese repräsentiert die Position des Eintrags in der alphabetischen Reihenfolge und damit die Zahl, die der Routine PlayScore() übergeben werden muß, um den Score wiederzugeben. Um die Auswahl eines Eintrags wieder rückgängig zu machen, klicken Sie diesen einfach noch einmal an. Er erscheint nun wieder in normaler Schrift und die Zahl davor verschwindet, auch werden die Ordnungsnummern der nachfolgenden angewählten Einträge um eins vermindert.

Die String-Gadgets

Obschon Sie den *SetPacker* auch allein mit der Maus bedienen können, ist es manchmal schneller, Datei- oder Verzeichnisnamen direkt über die Tastatur einzugeben. Klicken Sie dazu mit der Maus in den Bereich hinter dem Doppelpunkt, damit in dem gewünschten Gadget ein Cursor erscheint. Dann geben Sie den jeweiligen Namen ein und drücken anschließend <Return>, um die Gadgetfunktion zu aktivieren.

Es stehen folgende Stringgadgets zur Verfügung:

Das *Verzeichnis-Gadget*: Mit diesem Stringgadget kann man den Namen des Verzeichnisses, dessen Inhalt in der Filebox angezeigt werden soll, direkt eingeben. Gehen Sie dazu wie oben beschrieben vor. Beachten Sie bitte, daß bei einem Wechsel der Verzeichnisses sämtliche Auswahlmarkierungen gelöscht werden.

Das *Score-Name*-Gadget: Wenn z.B. ein Verzeichnis so lang ist, daß Sie es nicht von Hand durchsuchen möchten, können Sie die Namen der gewünschten Scores auch mit diesem Gadget eingeben. Wenn Sie den Namen korrekt getippt haben und <Return> drücken, erscheint der entsprechende Eintrag invers und ist somit für die Übertragung ins Set vorgemerkt. (Natürlich können Sie diese Markierung mit der Maus wieder löschen.)

Das *Instr.Verz.*-Gadget: In diesem Gadget geben Sie dem SetPacker den Namen des Verzeichnisses an, in dem nach den benötigten Instrument-Dateien gesucht wird. Es ist das einzige Stringgadget, dessen Inhalt korrekt initialisiert sein muß, da Sie sonst Fehlermeldungen wegen fehlender Instrumente erhalten.

Das *Ziel-File*-Gadget: Hier können Sie den Namen eingeben, unter dem das Set, das Sie gerade erstellen, abgespeichtert werden soll. Ihrer Eingabe wird, sollte es noch nicht darin enthalten sein, das Suffix ».sset« angefügt, um eine Verwechslung mit anderen Dateinamen zu verhindern. Wenn Sie dieses Feld leerlassen, wird Ihr Set einzig und allein unter der Endung ».sset« abgespeichert.

Die Buttons

Der *SetPacker* hat noch vier weitere Bedienungselemente, welche wie Schaltknöpfe, engl. Buttons, aussehen und deren Anklicken jeweils eine bestimmte Funktion auslösen kann. Diese wären im einzelnen:

Der *OK*-Button: Die Betätigung dieses Buttons beendet den Auswahlvorgang und startet die Erstellung der Set-Datei, worauf das Programm beendet wird.

Der *GERÄT*-Button: Mit diesem Button können Sie sich eine Liste der zur Verfügung stehenden Geräte in der Filebox anzeigen lassen, d.h., die in den Laufwerken befindlichen Disketten, angeschlossene Festplatten, Ramdisks etc. Sie müssen nur den entsprechenden Eintrag doppelklicken, um in das Root-Directory des gewünschten Geräts zu gelangen.

Der *Ü. VERZ.*-Button: Mit diesem Button gelangen Sie in das dem akuellen übergeordnete Dateiverzeichnis.

Der *ABBRUCH*-Button: Die Betätigung dieses Buttons beendet das Programm, ohne daß eine Set-Datei erzeugt wird.

4.3.3 Die Zusammenstellung eines Score-Sets

Direkt nach dem Start von *SetPacker* ist zunächst kein Verzeichnis angewählt, d.h., die Filebox ist leer. Geben Sie nun die Diskette, auf welcher sich Ihre Scores befinden, in eines der Laufwerke und drücken nach einiger Zeit den GERÄT-Button. Jetzt sollte unter anderem der Name Ihrer Diskette in der Geräteliste erscheinen. Öffnen Sie sie durch einen Doppelklick und wählen Sie Ihr Score-Verzeichnis aus. Sie können natürlich auch mit Hilfe des Verzeichnis-Gadgets direkt dorthingelangen.

Haben Sie nun die Namen Ihrer Scores vor sich, wählen Sie die gewünschten aus wie oben beschrieben und notieren sich bitte die zu den betreffenden Namen erscheinenden Nummern, damit Sie diese bequem beim Aufruf von PlayScore() verwenden können. Sodann klicken Sie auf den OK-Button, um die Erstellung der Set-Datei zu starten.

Da zur Erzeugung eines Sets sämtliche Dateien zuerst in den Arbeitsspeicher geladen werden müssen, ist der Ihnen zur Verfügung stehende Speicherplatz die einzige Begrenzung für die Anzahl der kombinierbaren Effekte. Speziell wenn diese viele verschiedene gesamplete Sounds beinhalten, kann es zu der Meldung »Couldn't alloc big buffer« kommen, was Speichermangel anzeigt. Booten Sie in diesem Fall am besten neu und halten die Ausführung der Startup-Sequence an, indem Sie CTRL-d gedrückthalten. Starten Sie den SetPacker von dem nun freigewordenen CLI aus und wiederholen Sie den Auswahlvorgang. Noch ein Tip: das Ausstecken eines evtl. Zweitlaufwerks bringt ebenfalls ca. 20 Kbyte!

4.4 SMUSPlayer-Library-Funktionsbeschreibung

LoadScore

Syntax: *erfolg = LoadScore(ScoreName, InstDir);*

(D0) *(A0)* *(A1)*

Beschreibung:
Stellt der Library einen SMUS-Score zur Wiedergabe durch PlayScore bereit.

Parameter:
ScoreName: Zeiger auf den Pfadnamen der Scoredatei, die geladen werden soll.

InstDir: Zeiger auf den Pfadnamen des Dateiverzeichnisses, in welchem nach den benötigten Instrumentdateien gesucht wird.

Ergebnis:
erfolg: ist 0, falls die Funktion ausgeführt werden konnte, –1 wenn ein Fehler aufgetreten ist.

C-Datentypen:

```
int erfolg;
char *ScoreName;
char *InstDir;
```

Kommentar:
siehe PlayScore, Stop, UnLoadScore

UnLoadScore

Syntax: *UnLoadScore();*

Beschreibung:
Gibt den Speicher, der von einem durch LoadScore geladenen Score (nebst Instrumenten) belegt wird, wieder frei.

Parameter:
keine Parameter

Ergebnis:
kein Ergebnis

Kommentar:
siehe LoadScore

LoadSet

Syntax: *erfolg = LoadSet(SetFileName);*
(D0) *(A0)*

Beschreibung:
Stellt der Library ein Set von SMUS-Scores zur Wiedergabe durch PlayScore bereit.

Parameter:
SetFileName: Zeiger auf den Pfadnamen des Sets, welches geladen werden soll.

Ergebnis:
erfolg: ist 0, falls die Funktion ausgeführt werden konnte, –1 wenn ein Fehler aufgetreten ist.

C-Datentypen:
```
int erfolg;
char *SetFileName;
```

Kommentar:
siehe PlayScore, Stop, UnLoadSet

UnLoadSet

Syntax: *UnLoadSet();*

Beschreibung:
Gibt den Speicher, der von einem durch LoadSet geladenen Score (nebst Instrumenten) belegt wird, wieder frei.

Parameter:
keine Parameter

Ergebnis:
kein Ergebnis

Kommentar:
siehe LoadSet

Taken from Amiga-Manuals-Website

PlayScore

Syntax: *erfolg = PlayScore(scorenum,repeats);*
D0 (D0) (D1)

Beschreibung:
Veranlaßt die Library, einen durch LoadScore oder LoadSet bereitgestelleten SMUS-Score wiederzugeben.

Parameter:
scorenum: Nummer des Scores im Set (entspricht der alphabetischen Reihenfolge der Filenamen der Scores, ist im Fenster des Setpacker-Programms ersichtlich). 0 bedeutet den durch LoadScore geladenen Score.
repeats: Anzahl der Wiederholungen des Scores. 0 bedeutet endlose Wiederholung.

Ergebnis:
erfolg: ist 0, falls die Funktion ausgeführt werden konnte, –1 wenn ein Fehler aufgetreten ist.

Kommentar:
siehe LoadScore, LoadSet, Stop

Stop

Syntax: *Stop();*

Beschreibung:
Veranlaßt die Library, die Wiedergabe zu stoppen. Aufruf während der Wiedergabe hält diese an und läßt alle momentan angeschlagenen Töne ausschwingen. Nochmaliger Aufruf (außerhalb der Wiedergabe) schaltet alle Kanäle sofort stumm.

Parameter:
Keine Parameter

Ergebnis:
Kein Ergebnis

Kommentar:
siehe PlayScore

LoadAux

Syntax: *erfolg = LoadAux(InstFileName);*
(D0) (A0)

Beschreibung:
Stellt der Library ein zusätzliches Instrument zur Verwendung durch PlayNote, PlayTone oder SetDefault bereit.

Taken from Amiga-Manuals-Website

Parameter:
InstFileName: Zeiger auf den Pfadnamen des Instruments, welches geladen werden soll.

Ergebnis:
erfolg: ist 0, falls die Funktion ausgeführt werden konnte, –1 wenn ein Fehler aufgetreten ist.

C-Datentypen:

```
int erfolg;
char *InstFileName;
```

Kommentar:
siehe PlayNote, PlayTone, SetDefault

SetDefault

Syntax: *erfolg = SetDefault(InstName);*
(D0) *(A0)*

Beschreibung:
Ermöglicht die Angabe eines Instruments, über das Tracks ohne definiertes Instrument wiedergegeben werden sollen.

Parameter:
InstName: Zeiger auf den Namen des Instruments, ohne Extension ».instr« o.Ä. 0 bedeutet Aux-Instrument.

Ergebnis:
erfolg: ist 0, falls die Funktion ausgeführt werden konnte, –1 wenn ein Fehler aufgetreten ist.

C-Datentypen:

```
int erfolg;
char *InstName;
```

Kommentar:
siehe FindInstr

PlayNote

Syntax: *voice = PlayNote(SPNote);*
(D0) *(A0)*

Beschreibung:
Ermöglicht die Wiedergabe eines Instruments mit einer bestimmten Tonhöhe und Tondauer.

Parameter:
SPNote: Zeiger auf eine SPNote-Struktur (zu den in dieser Struktur enthaltenen Prametern siehe 4.1.2.2).

Taken from Amiga-Manuals-Website

Ergebnis:
voice: Nummer des Kanals (1 – 4), falls die Funktion ausgeführt werden konnte, –1 wenn ein Fehler aufgetreten ist.

C-Datentypen:
```
int voice;
struct SPNote SPNote;
```

Kommentar:
siehe FindInstr

PlayTone

Syntax: *voice = PlayTone(SPNote);*
(D0) *(A0)*

Beschreibung:
Ermöglicht die Wiedergabe eines Instruments mit einer bestimmten Tonhöhe. Die Wiedergabe muß mit ReleaseTone wieder abgeschaltet werden.

Parameter:
SPNote: Zeiger auf eine SPNote-Struktur (zu den in dieser Struktur enthaltenen Prametern siehe 4.1.2.2).

Ergebnis:
voice: Nummer des Kanals (1–4), falls die Funktion ausgeführt werden konnte, –1 wenn ein Fehler aufgetreten ist.

C-Datentypen:
```
int voice;
struct SPNote SPNote;
```

Kommentar:
siehe FindInstr, ReleaseTone

ReleaseTone

Syntax: *ReleaseTone(voice);*
(D0)

Beschreibung:
Schaltet die Wiedergabe eines mit PlayTone gespielten Instruments ab. Kann auch zur vorzeitigen Beendigung einer von PlayNote oder von der Score-Wiedergabe gespielten Note verwendet werden.

Parameter:
voice: Nummer des Kanals, der abgeschaltet werden soll.

Ergebnis:
kein Ergebnis

Taken from Amiga-Manuals-Website

C-Datentypen:
```
int voice;
```

Kommentar:
siehe PlayTone

FindInstr

Syntax: *instr = FindInstr(InstName);*
(A0)

Beschreibung:
Gibt die Adresse der !!!-Struktur des jeweiligen Instruments zurück.
Parameter:

InstName: Zeiger auf den Namen des Instruments, ohne Extension ".instr" o.Ä. 0 bedeutet Aux-Instrument.

Ergebnis:
instr: Zeiger auf !!! – Struktur, falls ein Instrument dieses Namens gefunden wurde, 0 (!) wenn ein Fehler aufgetreten ist.

C-Datentypen:
```
struct !!! instr;
char *InstName;
```

Kommentar:
siehe LoadAux

SetReturnSig

Syntax; *SetReturnSig(SigMask)*
(D0)

Beschreibung:
Ermöglicht das Senden der in SigMask angegebenen Signale an den aufrufenden Task, ausgelöst durch das Ende (oder die Wiederholung) eines Scores.

Parameter:
Sigmask: Signalmaske, die an den Aufrufer gesendet werden soll. 0 = keine Signale (Abschalten der Option).

Ergebnis:
kein Ergebnis

C-Datentypen:
```
long SigMask;
```

Kommentar:
siehe PlayScore

Taken from Amiga-Manuals-Website

4.5 Die Demoprogramme

4.5.1 Assembler: der CLI-Befehl Play

Im Assembler-Verzeichnis Ihrer SMUSPlayer-Systemdisk befinden sich unter anderem zwei kleine Demoprogramme, die CLI-Befehle *Play* und *Stop*. Mit diesen können Sie die Wiedergabe eines SMUS-Scores starten und wieder beenden. Für Play bedienen Sie sich folgender Syntax:

»play <Score-Name> INSTRUMENT <Instr.-Verzeichnis>«

Wenn Sie das Schlüsselwort und den Verzeichnisnamen weglassen, so nimmt Play an, daß die Instrumente im Verzeichnis :Instruments zu finden sind (also im Verzeichnis Instruments in der Rootdirectory des akuellen Geräts).

Um die Wiedergabe anzuhalten, geben Sie einfach »stop« ein.

Beispiel: Sie möchten auf Ihrer Sonix-Sytemdiskette das Lied "Axel F" abspielen. Legen Sie sie z.B. in Laufwerk df1: und tippen Sie:

»play "df1:Scores/Axel F" INSTRUMENT df1:Instruments«

Sie können dies jetzt beliebig oft mit anderen oder demselben Score wiederholen (Play prüft allerdings nicht, ob sich ein angegebener Score schon im Speicher befindet), wobei die Wiedergabe für die Dauer des Zugriffs unterbrochen wird. Wenn Sie *Stop* eingeben, so wird die Library zwangsweise geschlossen und der belegte Speicher freigegeben. Dies ist zwar eine etwas unorthodoxe Methode, mit einer Library umzugehen, stellt aber eine einfache Möglichkeit dar, vom CLI aus einen Score im Hintergrund abspielen zu lassen. Verwenden Sie diese Befehle also bitte nicht, wenn schon in einem anderen Zusammenhang auf die Library zugegriffen wird!

Bemerkungen zum Listing:

Der hier verwendete CLI-Parser, der die Kommandozeile als Feld von Zeichenketten, ähnlich der Variablen *argv* in C, zur Verfügung stellt, kann vom Käufer (!) frei zur Erstellung eigener Programme verwendet werden.

Das Listing play.asm

```
; play.asm



        INCDIR      ":include/"

        Include     'exec/exec_lib.i'
        include     'asmsupp.i'
        include     'smusplayer.i'


run:
        move.l      #SRCBuf,SRCPointer
```

```
        subq    #1,d0                   ; Parms holen
        beq     noparms

search
        cmp.b   #$a,(a0)
        beq     noparms
        cmp.b   #$20,(a0)+
        bne     found
        dbra    d0,search
        bra     noparms

found
        lea     SongBuf(PC),a1
        lea     argv(PC),a3

loop1a
        move.l  a1,(a3)+                ; Parameter-Wortanfang in argv
        addq.w  #1,argc
        move.b  -(a0),d0                ; Schritt zurück
        addq    #1,a0                   ; und wieder vor
        cmpi.b  #'"',d0                 ; Anführungszeichen?
        beq.s   quoted                  ; ja ->
        move.b  d0,(a1)+

loop1
        move.b  (a0)+,d0
        cmpi.b  #$20,d0
        bne     notSt1

loop2a
        move.b  #0,(a1)+                ; Parameter durch 0 trennen

loop2
        cmpi.b  #$a,(a0)
        beq     astart
        cmpi.b  #$20,(a0)+
        beq     loop2
        bra.s   loop1a

notSt1:
        cmpi.b  #$a,d0
        bne     notSt2
        move.b  #0,(a1)

astart:
        bra     OpenLib

notSt2:
        move.b  d0,(a1)+                ; gesamten String kopieren
        bra     loop1                   ; bis Leerzeichen

noparms:
        rts
```

Taken from Amiga-Manuals-Website

```
quoted
        move.b      (a0)+,d0                    ; alle Zeichen kopieren bis
        cmpi.b      #$a,d0                      ; Return
        beq.s       astart
        cmpi.b      #'"',d0                     ; oder     Anf.zeichen
        beq.s       loop2a
        move.b      d0,(a1)+
        bra.s       quoted

OpenLib:
        lea         LibName(PC),a1              ; SMUSPlayer öffnen
        clr.l       d0
        movea.l     4,a6
        CALLSYS     OpenLibrary
        move.l      d0,SMUSPlayerBase
        beq.s       noparms

INSTausw:
        cmpi.w      #3,argc
        blt         AEnd
        movea.l     argv+4,a0
        lea         KeyWord(PC),a1              ; Schlüsselwort vorhanden?

loop6:
        move.b      (a0)+,d0
        beq         OK1
        andi.b      #%11011111,d0
        cmp.b       (a1)+,d0
        bne         AEnd
        bra         loop6

OK1
        movea.l     argv+8(PC),a1               ; ja: InstDirName = 3. Parameter
        bra.s       A1End

AEnd:
        movea.l     DefaultInstDir(PC),a1       ; Score laden
A1End
        movea.l     argv(PC),a0
        LINKSYS     LoadScore,SMUSPlayerBase
        tst.l       d0
        bmi.s       PEnd
        moveq       #0,d0                       ; "LoadScore"-Score
        moveq       #1,d1                       ; einmal abspielen

        LINKSYS     PlayScore,SMUSPlayerBase    ; und abspielen

PEnd
        moveq       #0,d0                       ; Library soll erst mit "stop"
        rts                                     ; geschlossen werden

LibName:
        dc.b        'smusplayer.library',0
```

Taken from Amiga-Manuals-Website

```
KeyWord:
        dc.b        'INSTRUMENT',0

DefaultInstDir:
        dc.b        ':Instruments',0

EVEN

SMUSPlayerBase:
        ds.l        1

SongBuf:
        ds.b        100

MySong:
        ds.b        100

ScoreName:
        ds.l        1

InstDirName:
        ds.l        1

argv:
        ds.l        10                      ; Array aus Zeigern auf Parm.-Namen

argc:
        dc.w        0                       ; Anzahl Parameter

SRCPointer:
        dc.l        0

SRCBuf:
        ds.w        200

SRCBufEnd:
```

Das Listing stop.asm

```
;  stop.asm


 INCDIR  ":include/"

 include 'exec/libraries.i'
 include 'exec/exec_lib.i'
 include 'asmsupp.i'
 include 'smusplayer.i'
```

Taken from Amiga-Manuals-Website

```
        lea         LibName,a1
        moveq       #0,d0                       ; Version egal
        move.l      4,a6
        CALLSYS     OpenLibrary                 ; Library öffnen
        move.l      d0,SMUSPlayerBase
        beq.s       End

        movea.l     d0,a5
        move.w      #1,LIB_OPENCNT(a5)          ; zum Schließen zwingen

        movea.l     d0,a1
        CALLSYS     CloseLibrary

End
        rts

LibName:
      dc.b     'smusplayer.library',0

SMUSPlayerBase:
      ds.l     1
```

4.5.2 Beispiel in C

Um die Verwendung der Routinen *LoadSet* und *PlayScore* in der Praxis zu verdeutlichen, ist im Demos/CC-Verzeichnis Ihrer SMUSPlayer-Systemdisk ein Demoprogramm vorgesehen, welches ein simples Schlagzeug darstellt. Um es zu starten, öffnen Sie entweder die Demos-Schublade in der Workbench und klicken sein Icon an, oder Sie gehen in den CLI und tippen:

»AEP:Demos/Schlagzeug«

Das Programm öffnet ein Fenster, in dem mit verschiedenen Rhythmen- und Instrumentennamen gekennzeichnete Felder erscheinen. Wenn Sie nun in eines dieser Felder klicken, werden Sie einen Schlagzeugrhythmus bzw. das jeweilige Instrument aus dem Lautsprecher vernehmen.

Die Rhythmen und Instrumente sind Teil eines Sets, welches im Demos-Verzeichnis unter dem Namen »Schlagzeug.sset« abgespeichert ist. Die Scores, aus denen es zusammengesetzt ist, sind mit Sonix erstellt worden und bestehen jeweils nur aus einem kurzen Schlagzeugbeat, der ständig wiederholt wird. Ebenso kann man mit jedem anderen Instrument auf diese Weise beliebige Soundeffekte zusammenstellen, speziell wenn man in Besitz eines Digitalisierers und der entsprechenden Software ist.

Das Listing Schlagzeug.c

```
/*************************************************************************

     Schlagzeug.c (c) Nicolaus Wirsing 17.4.89 00:21

     All Rights reserved.

*************************************************************************/
```

Takeda Amiga Manuals-Website

```
#include <intuition/intuitionbase.h>
#include <graphics/gfxbase.h>
#include <intuition/intuition.h>
#include "smusplayer_lib.h"


#define XOFFS       10          /*** Konstanten für Window und "Buttons" ***/
#define YOFFS       20
#define STCLENGTH   8
#define STPLENGTH   (STCLENGTH*CHSIZE)
#define CHSIZE      8
#define CHVERT      15
#define HINDENT(c)  ((int)((STCLENGTH-strlen(c))*CHSIZE/2))
#define VINDENT     4
#define BXOF        ((STCLENGTH+4)*CHSIZE)
#define BYOF        (CHVERT*3)
#define WLEFT       100
#define WTOP        50
#define WWIDTH      ((2*XOFFS)+3*(BXOF))
#define WHEIGTH     ((YOFFS)+3*(BYOF))
#define INSTS       3
#define RHYTHMS     3


struct BUTTON
{
     USHORT x1,y1;                                  /*** Button-Position ***/
     USHORT x2,y2;
     char *text;                                        /*** Button-Text ***/
};

struct BUTTON IButtons[INSTS] =                         /*** Instrumente ***/
{
     { XOFFS, YOFFS, XOFFS+STPLENGTH, YOFFS+CHVERT,"Bass"},
     { XOFFS+1*BXOF, YOFFS, XOFFS+STPLENGTH+1*BXOF, YOFFS+CHVERT, "Snare" },
     { XOFFS+2*BXOF, YOFFS, XOFFS+STPLENGTH+2*BXOF, YOFFS+CHVERT, "HighHat"}
};

struct BUTTON RButtons[RHYTHMS] =                   /*** Begleitrhythmen ***/
{
     { XOFFS, YOFFS+BYOF, XOFFS+STPLENGTH, YOFFS+CHVERT+BYOF, "Beat"},
     { XOFFS+1*BXOF, YOFFS+BYOF, XOFFS+STPLENGTH+1*BXOF, YOFFS+CHVERT+BYOF,
     "Break"},
     { XOFFS+2*BXOF, YOFFS+BYOF, XOFFS+STPLENGTH+2*BXOF, YOFFS+CHVERT+BYOF,
     "Heavy"}
};

struct BUTTON QButton =                                 /*** Stop-Button ***/

     { XOFFS+1*BXOF, YOFFS+2*BYOF, XOFFS+STPLENGTH+1*BXOF,
                                   YOFFS+CHVERT+2*BYOF, "Quiet"};
```

Taken from Amiga-Manuals-Website

```
static char *SetName = "Schlagzeug.sset";          /***   akt. Directory!   ***/

static char *INames[] = {                          /*** Instr-Namen im Set ***/
     "BassDrum",
     "SnareDrum",
     "HighHat"
};

int lastNote = 0;                            /*** zuletzt gespieltes Instr. ***/

struct NewWindow MyNewWindow = {
     WLEFT,WTOP,
     WWIDTH,WHEIGTH,
     -1,-1,
     REFRESHWINDOW|MOUSEBUTTONS|CLOSEWINDOW,
     WINDOWDRAG|WINDOWDEPTH|WINDOWCLOSE|SIMPLE_REFRESH|ACTIVATE,
     NULL,NULL,
     (UBYTE *)"Schlagzeug",
     NULL,NULL,
     0,0,0,0,
     WBENCHSCREEN
};

struct IntuitionBase  *IntuitionBase;
struct GfxBase        *GfxBase;
struct SMUSPlayerBase *SMUSPlayerBase;
struct Window         *MyWindow;
struct IntuiMessage   *msg;


main()
{
   int CloseFlag = 0;

                                        /*** Graphics-Library öffnen ***/
   if(!(GfxBase = (struct GfxBase *)OpenLibrary("graphics.library",0)))
   {
         printf("\nKonnte graphics.library nicht öffnen !\n");
         goto Ende;
   }

                                        /*** Intuition-Library öffnen ***/
   if(!(IntuitionBase = (struct IntuitionBase *)
                  OpenLibrary("intuition.library",0)))
   {
         printf("\nKonnte intuition.library nicht öffnen !");
         goto Ende;
   }

                                        /*** SMUSPlayer-Library öffnen ***/
   if(!(SMUSPlayerBase = (struct SMUSPlayerBase *)
                        OpenLibrary("smusplayer.library",0)))
   {
```

Taken from Amiga-Manuals-Website

```
        printf("\nKonnte SMUSPlayer.library nicht öffnen !");
        goto Ende;
    }

                                        /*** "Schlagzeug.sset" laden ***/
    if(LoadSet(SetName))
    {
        printf("\nKonnte Set %s nicht laden !",SetName);
        goto Ende;
    }

            /*** Window öffnen und initialisieren (Buttons zeichnen) ***/
    if(!(MyWindow = (struct Window *)OpenWindow(&MyNewWindow)))
    {
        printf("\nKonnte Window nicht öffnen !");
        goto Ende;
    }

    DrawWindowContents();                /***  Window-inhalt zeichnen   ***/

    while(!CloseFlag)
    {                                   /*** Bis Click auf Close-Gadget ***/
        Wait(1<<MyWindow->UserPort->mp_SigBit);

        while(msg = (struct IntuiMessage *)GetMsg(MyWindow->UserPort))
        {
            switch(msg->Class)
            {
                case MOUSEBUTTONS:{
                        ProcessMouse(msg->MouseX,msg->MouseY, msg->Code);
                        break;
                    }
                case REFRESHWINDOW:{
                        BeginRefresh(MyWindow);
                        DrawWindowContents();
                        EndRefresh(MyWindow, 1);
                        break;
                    }
                case CLOSEWINDOW:{
                        CloseFlag = 1;
                        break;
                    }
            }
            ReplyMsg(msg);
        }
    }

Ende:                                                   /*** Aufräumen ***/
    if(MyWindow)        CloseWindow(MyWindow);
    if(GfxBase)         CloseLibrary(GfxBase);
    if(IntuitionBase)   CloseLibrary(IntuitionBase);
    if(SMUSPlayerBase)  CloseLibrary(SMUSPlayerBase);
    exit(1);
}
```

Taken from Amiga-Manuals-Website

```
                                                    /*** Fensterinhalt zeichnen ***/
DrawWindowContents()
{
   int but;

   if(MyWindow)
   {
        for(but=0; but<INSTS    ; but++)     DrawButton(IButtons, but, 2);
        for(but=0; but<RHYTHMS ; but++)     DrawButton(RButtons, but, 3);
        DrawButton(&QButton,0,0);
     }
}


                                                    /*** Button zeichnen ***/
DrawButton(ButVect,n,color)
struct BUTTON *ButVect;
int n, color;
{
   struct RastPort *rp;

   rp = MyWindow->RPort;
   SetAPen(rp,color);
   RectFill(rp, ButVect[n].x1, ButVect[n].y1,
                  ButVect[n].x2, ButVect[n].y2);
   SetAPen(rp,1);
   SetBPen(rp,color);
   Move(rp, HINDENT(ButVect[n].text)+ButVect[n].x1,
             VINDENT+CHSIZE+ButVect[n].y1);
   Text(rp, ButVect[n].text, strlen(ButVect[n].text));
}


                                /*** Hier werden die Maus-Clicks behandelt ***/
int ProcessMouse(x, y, code)
int x, y, code;
{
   int but;
   struct SPNote SPNote;

   if(code == SELECTDOWN)
   {                                                /*** Maus gedrückt? ***/
        for(but=0;but<INSTS;but++)
        {
                                                    /*** Instr.-Button? ***/
             if( x>=IButtons[but].x1 && x<=IButtons[but].x2 &&
                 y>=IButtons[but].y1 && y<=IButtons[but].y2 )
             {
                                        /*** SNote- Struktur vorbereiten ***/
                 SPNote.iName = INames[but];
                 SPNote.SNote = 0x3000;
```

Taken from Amiga-Manuals-Website

```
                SPNote.vol   = SP_MAXVOL;
                SPNote.voice = SP_DYNAMICVOICE;
                SPNote.pri   = SP_PLAYERPRI+1;
                lastNote = PlayTone(&SPNote);
                break;
            }
            else
                                                /*** Rhythm-Button? ***/
                if( x>=RButtons[but].x1 && x<=RButtons[but].x2 &&
                    y>=RButtons[but].y1 && y<=RButtons[but].y2 )
                {
                    PlayScore(but+1,0);
                    break;
                }
            else
                                                /***  Stop-Button?  ***/
                if( x>=QButton.x1 && x<=QButton.x2 &&
                    y>=QButton.y1 && y<=QButton.y2 )
                {
                    Stop();
                    break;
                }
        }
    }

    if((code == SELECTUP) && lastNote)
    {                                           /***     Maus losgelassen?   ***/
        ReleaseTone(lastNote);                  /*** Ja: letzten Ton stoppen ***/
        lastNote = 0;
    }

    return();
}
```

Auf der mitgelieferten Systemdiskette befinden sich im Verzeichnis AEP:Demos/CC noch zwei weitere Demo-Programme. NoteDemo.c verdeutlicht den Gebrauch der SPNote-Struktur zur Tonausgabe über das audio.device. Das Programm lädt nach dem Start ein Instrument Namens SynthSound aus dem Verzeichnis INSTRUMENTS: und spielt es sodann in Achteln zwei Oktaven ab dem mittleren C (MIDI-Nummer 60) hoch und danach in Sechzehnteln wieder herunter. Die Tonausgabe geschieht mit den Funktionen PlayNote(), PlayTone() und ReleaseTone() der SMUSPlayer.library. SyncDemo.c verdeutlicht die Synchronisation eines Scores mit dem Programm. Nach dem Start über das CLI wird das schon bekannte »Schlagzzeug.sset« geladen, welches ja aus drei kurzen Schlagzeug-Pattern besteht. Diese werden nun nacheinander abgespielt, wobei die SMUSPlayer.library nach jedem Abspielen ein Signal an das Programm sendet, wonach veranlaßt wird, das nächste Pattern abzuspielen. Anhand dieses Beispiels wird der Umgang mit AllocSignal(), SetReturnSignal(), Wait() und FreeSignal() dargestellt. Beide Programme müssen über das CLI mit folgendem Aufruf gestartet werden: »AEP:Demos/NoteDemo«, bzw. »AEP:Demos/SyncDemo«.

Tandy Computer-Manuals-Website

Das Listing NoteDemo.c

```
/*****************************************************/
/***                    NoteDemo.c                 ***/
/***                © Ilias Vassiliou              ***/
/***                    31.08.1990                 ***/
/***                   AZTEC-C V3.6                ***/
/***  Compile : cc +l NoteDemo.c                   ***/
/***  Link    : ln NoteDemo.o SmusPlayer_lib.o -lc32 ***/
/*****************************************************/


#include "smusplayer_lib.h"

struct SMUSPlayerBase *SMUSPlayerBase;

struct SPNote SPNote;
UWORD  NotenNummer=60,Note=0;  /***  NotenNummer=60 bedeutet mittleres C ***/

BYTE *Instrument = "INSTRUMENTS:SynthSound";
WORD voice;                                        /*** Ausgabe-Kanal ***/


VOID
CloseIt()                  /***  Library schließen (falls geöffnet !!!) ***/
{
     if(SMUSPlayerBase!=NULL) CloseLibrary(SMUSPlayerBase);
     exit(0);
}

VOID
main()
{
     if(!(SMUSPlayerBase = (struct SMUSPlayerBase *)   /*** Library öffnen ***/
                              OpenLibrary("smusplayer.library",0)))
   {
          puts("\nKonnte smusplayer.library nicht öffnen !!!");
          CloseIt();
     }

     if( LoadAux(Instrument) == -1 )   /*** "INSTRUMENTS:SynthSound" laden ***/
   {
      puts("\nKonnte Instrument nicht laden !!!");
      CloseIt();
   }
                    /*** Default Instrument = 0 (Aux-Instrument) setzen ***/
     if( SetDefault((BYTE *)0) == -1)
   {
      puts("\nKonnte Default-Instrument nicht setzen !!!");
      CloseIt();
   }
```

Taken from Amiga-Manuals-Website

```
SPNote.iName    =  SP_DEFAULTINAME; /***       Aux-Instrument          ***/
SPNote.vol      =  0xff;            /***       Volle Lautstärke        ***/
SPNote.voice    =  SP_DYNAMICVOICE; /*** Freie Wahl des Ausgabe-Kanals ***/
SPNote.pri      =  SP_PLAYERPRI+1;  /***       Priorität setzen        ***/


  /***  Die Variable SNote ist in dem Include-File : "iff/smus.h"  ***/
  /***  als 16-Bit-Feld definiert; davon geben die ersten 8 Bit    ***/
  /***  die MIDI-Noten-Nummer an (0-127), das 9. Bit legt fest ob  ***/
  /***  die Note zu einem Akkord gehört, das 10., ob die Note gebun-  ***/
  /***  den ist, das 11. und 12. Bit geben an, ob es sich um eine  ***/
  /***  Triole, Quintole oder Septole handelt, das dreizehnte Bit  ***/
  /***  wird bei punktierten Noten gesetzt und die letzten drei Bits  ***/
  /***  geben die Länge der Note an (ganze, halbe, Viertel, ... ,  ***/
  /***  1/128tel).                                                 ***/
  /***  Es ist am besten, wenn man sich das Include-File sehr genau  ***/
  /***  ansieht - dort sind auch einige Standard-Werte definiert,  ***/
  /***  die im folgenden verwendet werden (noteD8,noteD16) !       ***/

  /***  Wichtig zu wissen ist, daß bei der Ausgabe einer Note mit  ***/
  /***  PlayNote() nicht gewartet wird, bis die Note mit dem defi-  ***/
  /***  nierten Notenwert (8tel,16tel) abgeklungen ist.            ***/
  /***  Aus diesem Grunde ist in der While-Schleife ein Delay(10)  ***/
  /***  eingebaut, um die gespielten Noten mit der jeweiligen Noten  ***/
  /***  dauer überhaupt wahrnehmen zu können.                      ***/


  /***  Der folgende Programmteil spielt ab dem mittleren C ( MIDI-  ***/
  /***  Note 60 ) zwei Oktaven hoch alle Halbtöne in Achteln und   ***/
  /***  danach zwei Oktaven wieder zurück - diesmal in Sechzehnteln  ***/

while( (NotenNummer++) < 85 )  /***        Hier geht's hoch          ***/
{
   Note  = (NotenNummer<<8);   /***     MIDI-NotenNummer setzen      ***/
   Note |= noteD8;             /*** Notendauer setzen noteD8 = achtel ***/
   SPNote.SNote = Note;        /*** Wert der SPNote-Struktur zuweisen ***/
   voice = PlayNote(&SPNote);  /***  Note nun ausgeben voice = Kanal  ***/
   Delay(10);                  /***       1/5-Sekunde Verzögerung     ***/
}

while( (NotenNummer--) > 62 )  /***     Und hier wieder 'runter      ***/
{
   Note  = (NotenNummer<<8);
   Note |= noteD16;            /*** Notendauer setzen noteD16 = 16tel ***/
   SPNote.SNote = Note;
   voice = PlayNote(&SPNote);
   Delay(10);
}

Delay(50);                     /***            Kurz warten          ***/
```

Taken from Amiga-Manuals Website

```
   SPNote.SNote = (60<<8);          /*** Note C (60) der Struktur zuweisen ***/
   voice = PlayTone(&SPNote);       /***  Note nun ausgeben voice = Kanal  ***/
   Delay(100);                      /***       2 Sekunden Verzögerung      ***/
   ReleaseTone(voice);              /***       Kanal wieder freigeben      ***/


   CloseIt();
}
```

Das Listing SyncDemo.c

```
/*****************************************************/
/***                    SyncDemo.c                 ***/
/***              Copyright Ilias Vassiliou        ***/
/***                    31.08.1990                 ***/
/***                   AZTEC-C V3.6                ***/
/***  Compile : cc +l SyncDemo.c                   ***/
/***  Link    : ln SyncDemo.o SmusPlayer_lib.o  -lc32 ***/
/*****************************************************/


#include "iff/smus.h"
#include "smusplayer_lib.h"

struct SMUSPlayerBase *SMUSPlayerBase=NULL;

BYTE   *SetName = "Schlagzeug.sset";      /*** Name des zu ladenden Sets ***/

BYTE   Signalnummer;                          /*** selbsterklärend ***/
ULONG  Signale;


VOID
CloseIt()     /***  Library schließen (falls geöffnet !) und Programmende ***/
{
     if(SMUSPlayerBase!=NULL) CloseLibrary(SMUSPlayerBase);
     exit(0);
}


VOID
main()                                                /*** Hauptprogramm ***/
{
                                        /*** smusplayer.library öffnen ***/
   if(!(SMUSPlayerBase = (struct SMUSPlayerBase *)
                                 OpenLibrary("smusplayer.library",0L)))
   {
```

Taken from Amiga-Manuals-Website

```
        printf("\nKonnte smusplayer.library nicht öffnen !!!\n");
        CloseIt();
    }

    if(LoadSet(SetName)==-1)                                /*** Set laden ***/
    {
        printf("\nKonnte Set '%s' nicht laden !!!\n",SetName);
        CloseIt();
    }
                                            /*** Signalnummer ermitteln ***/
    if( (Signalnummer = (BYTE)AllocSignal(-1L)) == -1)
    {
       printf("\nAlle Signale schon besetzt - kein Demo möglich !!!\n");
       CloseIt();
    }
    else
       SetReturnSig((LONG)1<<Signalnummer);     /***   Sync-Signal setzen  ***/

    printf("\nSignalnummer = %d",Signalnummer);/*** Signalnummer ausgeben ***/


    PlayScore(1,0);                             /***  Pattern 1 abspielen  ***/
    printf("\nNach PlayScore 1.");

                                /*** Auf Synchronisations-Signal warten ***/
    Signale = Wait((ULONG)1<<Signalnummer);
    printf("\nEnde 1 erreicht, Signal = %ld.",Signale);


    PlayScore(2,0);                             /*** Pattern 2 abspielen ***/
    printf("\nNach PlayScore 2.");

                                /*** Auf Synchronisations-Signal warten ***/
    Signale = Wait((ULONG)1<<Signalnummer);
    printf("\nEnde 2 erreicht, Signal = %ld.",Signale);


    PlayScore(3,0);                             /*** Pattern 3 abspielen ***/
    printf("\nNach PlayScore 3.");

                                /*** Auf Synchronisations-Signal warten ***/
    Signale = Wait((ULONG)1<<Signalnummer);
    printf("\nEnde 3 erreicht, Signal = %ld.\n",Signale);

    FreeSignal((ULONG)Signalnummer);      /*** Signal wieder freigeben !!! ***/

    UnLoadSet();                /*** Speicher des Sets wieder freigeben ***/
                                /*** - ist hier nicht unbedingt nötig   ***/

    CloseIt();
}
```

Taken from Amiga-Manuals-Website

4.6 Installation

4.6.1 Anlegen einer Sicherungskopie

Wie von allen Ihren Originaldisketten, so sollten Sie auch von Ihrem AEP-Exemplar zuerst eine Sicherungskopie anlegen. Booten Sie hierzu mit dem Original und legen Ihre Zieldiskette in Laufwerk »df1:«. Dann geben Sie ein:

»diskcopy df0: to df1: <RETURN>«

Falls Sie nur ein Laufwerk besitzen sollten, so tippen Sie bitte:

»diskcopy df0: to df0: <RETURN>«

Entfernen Sie nun das Original aus dem Laufwerk und bewahren es an einem sicheren Ort auf, d.h. trocken und möglichst weit entfernt von magnetischen Störquellen, wie z.B. Transformatoren in elektrischen Geräten, Lautsprecher etc. Arbeiten Sie immer nur mit Ihrer Sicherungskopie und halten Sie Ihr Original als »eiserne Reserve« bereit, denn es ist logischerweise wesentlich wahrscheinlicher, nach häufigem Laden und Speichern einen Fehler auf einer Diskette zu erhalten als nach bloßem Lagern.

4.6.2 Installation der SMUSPlayer-Library

Das Amiga-Betriebssystem sucht nach externen Libraries stets im logischen Laufwerk »libs:«, welches normalerweise dem Verzeichnis »libs« auf der Bootdiskette zugeordnet ist. Zur Installation von SMUSPlayer kopieren Sie diesen zuerst auf Ihre bei der Programmentwicklung verwendete Arbeitsdiskette. Geben Sie dazu z.B.

»copy AEP:libs/smusplayer.library MeineDisk:libs«

ein , wobei MeineDisk dem Namen Ihrer Diskette entsprechen muß. Booten Sie mit dieser sodann wie gewohnt, um Ihre normale Arbeitsumgebung herzustellen. Ab jetzt können Sie selbst auf die Routinen von SMUSPlayer zugreifen.

4.6.3 Programmiersprachen-spezifische Installationen

Um die Library-Funktionen von der von Ihnen verwendeten Programmiersprache aus aufrufen zu können, müssen Sie Ihrer Arbeitsumgebung noch einige Elemente hinzufügen. Folgen Sie dazu bitte den Anweisungen in dem betreffenden Abschnitt.

Assembler

Um auf die Library in Assembler zugreifen zu können, müssen Sie Ihrem Übersetzer die Werte für die Offsets derselben mitteilen. Hierzu kopieren Sie die Dateien SMUSPlayer_Lib.i und asmsupp.i, die sich im Verzeichnis AEP:SDK/ASSEMBLER/Include befinden, in das INCLUDE-Verzeichnis Ihres Assemblers, z.B. mit den Befehlszeilen:

»CD AEP:SDK/ASSEMBLER«
»COPY SMUSPlayer_lib.i MeineDisk:INCLUDE«
»COPY asmsupp.i MeineDisk:INCLUDE«

Die Datei *asmsupp.i* enthält einige Makros, u.a die Makros CALLSYS und LINKSYS, welche eine bequeme Aufrufmöglichkeit für die Funktionen aller Libraries darstellen. Jetzt müssen Sie nur noch in Ihr Assembler-Programm

»INCLUDE "SMUSPlayer_Lib.i"«

einfügen, um die Routinen der Library zu verwenden. Näheres zu den Aufrufkonventionen von Libraries in Assembler erfahren Sie in Kapitel 4.2.1.

C

Um die Library-Funktionen in C aufrufen zu können, führen Sie bitte folgende Schritte durch:

Kopieren Sie die Datei SMUSPlayer_lib.h, die sich im Verzeichnis AEP:SDK/CC/Include befindet, in das INCLUDE-Verzeichnis Ihrer C-Arbeitsdiskette, z.B mit den Befehlszeilen:

»CD AEP:SDK/CC/INCLUDE«
»COPY SMUSPlayer_lib.h MeineDisk:INCLUDE«

Die Interfaceroutinen, welche die Aufrufkonventionen von C denen der Library anpassen, befinden sich in dem Objektfile SMUSPlayer_lib.o, ebenfalls im CC-Verzeichnis. Kopieren Sie diese Datei in dasjenige Verzeichnis, in welchem Ihr Linker nach den angegebenen Objektfiles sucht. Fügen Sie seinen Namen von jetzt an immer in das Linkerkommando ein (allerdings nicht an erster Stelle), um den Linker die richtigen Verbindungen zum System herstellen zu lassen.

Eine weitere Möglichkeit, die Interfaceroutinen einzubinden, besteht in der Umwandlung derselben in eine sogenannte »linked library«, einem Äquivalent zu jenen ».lib« -files, in welchen die von C zur Verfügung gestellten Funktionen untergebracht sind. Um das Objektfile SMUSPlayer_lib.o in eine solche »linked library« umzuwandeln, befolgen Sie bitte die diesbezüglichen Anweisungen im Handbuch Ihres C-Compilers.

Näheres zu den Aufrufkonventionen in C erfahren Sie im Abschnitt 4.2.2

Basic

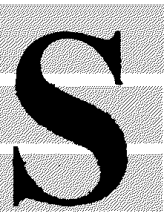
Um die Library-Funktionen in Basic aufrufen zu können, benötigt der Interpreter deren Offsets in Form von sog. ».bmap« -Dateien, nach denen er sowohl im aktuellen Dateiverzeichnis als auch im Verzeichnis »libs:« sucht. Damit Sie diese nicht erst aus der (ebenfalls mitgelieferten) ».FD«-Datei erzeugen müssen, ist eine solche bereits im Basic-Verzeichnis der Systemdiskette enthalten. Kopieren Sie sie mit dem Befehl

»copy AEP:SDK/BASIC/SMUSPlayer.bmap MeineDisk:libs«

auf Ihre Basic-Arbeitsdiskette, damit Sie beim Öffnen mit dem Library-Befehl von dort geholt werden kann.

Näheres zu den Aufrufkonventionen in Basic erfahren Sie im Kapitel 4.2.3.

Stichwortverzeichnis



A

A/D-Wandler 33, 35
Abtastrate 48, 56
Abtastung 33, 45
Active Sensing 168, 177
ADCMD_ALLOCATE 115
ADCMD_FINISH 120
ADCMD_FREE 117
ADCMD_LOCK 117
ADCMD_PERVOL 121
ADCMD_SETPREC 118
ADCMD_WAITCYCLE 121
Additive Synthese 222
ADKCON-Register 91
Adressenanpassung 256
ADSR-Werte 29
After Touch 163
Aftertouch polyphon 158
Agnus 62
Akkorde 271
Aliasing 47
Alignment 256
Amiga-Audio-Maschine 99
Amiga-Audiohardware 59
Amiga-Betriebssystem 107
Amplitude 16
Amplitudenänderung 235
Amplitudenfehler 38
Analoge Syntheseverfahren 221
Anblasverhalten 29
Anti-Aliasing-Filter 51
Assembler 324
Assembler-Programmierung 299
Asynchroner IO 183
ATAK-Chunk 286
Attack 30
Audio-Device 113
Audio-Interrupt 95
Audio-Zustandsdiagramm 98
Audioausgabe 113
AUDxLC-Register 70
AUDxLEN-Register 72
AUDxPER-Register 73
AUDxVOL-Register 73
Aufnahme 133, 249
Aufnahmerate 58
Aussteuerung 42, 134, 250
Auto-Locator 245
Automatentheorie 98

B

Backing-Track 141
Bandmaschine 242
Basic 325
Basic-Parameter 234
Basic-Programmierung 301
Binärzahlen 33
BODY-Chunk 287
Bouncing 246
Buttons 303

C

C 325
C-Programmierung 300
CAT-Chunk 261
CD-Player 131
Characters 256
Chorus 240

Chunk-Prinzip 257
Clipping-Verzerrung 44
CMD_FLUSH 120
CMD_READ 120
CMD_RESET 120
CMD_START 120
CMD_STOP 120
CMD_WRITE 118
Continue 168
Control/Mode Change 158
Cue-Locator 245
Customchips 60
Cycle 65

D

D/A-Wandler 55
Daten-Byte 155
Datentypen 256
DBX NR 244
Decay 30
Demoprogramme 310
Device-Funktionen 180
Device-I/O 111
Device-spezifische Kommandos 181
Devices 107
Dezibel 18
Digitale Verfahren 31
Digitaltechnik 15
Direkter Device-Einsprung 184
Distorsion 240
Distorsion-Synthese 223
DMA 60
DMA-Betrieb 70
DMA-gesteuerte Audioausgabe 99
DMA-Kontrolle 77
DMA-Slots 63
DMA-Timing 63
DMACON-Register 78
Dolby B/C 244
Doppler-Effekt 91
Double-Buffering 119
Drum-Sequencer 230
Dualsystem 34
Dynamik 139
Dynamikgewinn 42

E

EBU-Timecode 248
Echo 238
Effekte 237
Elektrische Instrumente 130
Endabmischung 251
EOX 167
Equalizer 239
Events 263
Expander 233

F

Fibonacci-Delta-Codierung 289
Filebox 302
FindInstr 296, 309
Flanging 240
FORM-8SVX-Typ-Definitionen 290
FORM-Chunk 259
FORM-Statement 260
Fourier-Analyse 21
Fourier-Reihen 23
Frequenz 19
Frequenz-Modulation 90
Frequenzgang 242
Frequenzmodulations-Synthese 224
FSK-Sync 247
Funktionen 179

G

Gehör 16
Gerätetreiber 107
Geräusch 16, 27
Granularrauschen 43

H

Hall 238
Haltphase 135

Taken from Amiga-Manuals-Website

Harmonische Schwingung 21
Hertz 20
Hüllkurve 29, 135

I

I/O-Request 111
Identifier 257
Idle State 61
IFF-File 260
IFF-FORM-Familie 263
IFF-Format 255
INS1-Chunk 268
Installation 186, 324
Instrumentänderung-Event 272
Instrumente 65, 134, 264
Instrumentenformate 281
INTENA-Register 82
Interface-Hardware 146
Interferenzen 21
Interrupt-gesteuerte Audioausgabe 99
Interrupts 81
INTREQ-Register 82
IOMIDI 177
IOM_TypeFlags 178

K

Kanalkombination 115
Kawai K1II 227
Keyboard 226
Keyboard.c 216
Keysigs 274
Klang 21
Klangdaten 64
Klangeffekt 89
Klangeigenschaften 242
Klangspektrum 21
Klangwiedergabe 64
Klick-Track 247
Kommandos 179
Komposition 138
Kompositionsprogramme 298
Kompression 239
Korg-A3-Multieffekt-Prozessor 241

L

Längen-Raster 137
Längenzähler 257
Lautstärke 16
Lautstärke setzen – Event 275
Lautstärke-Effekte 239
Lautstärke-Modulation 89
Lautstärkekorrektur 135
Lautstärkeregister 62
Lautstärkeverlauf 29
Leslie 91
Level-Parameter 235
LFO-Parameter 235
Library 293
Library-Programmierung 298
LIST-Chunk 262
LoadAux 296, 306
LoadScore 294, 304
LoadSet 295, 305
Local-on/off-Umschaltung 151
Loop-Instrumente 136
Looping 136
LSB 33, 155

M

Maskieren 43
Masterrecording 237
Mehrdeutigkeit 47
Mehrspur-Bandmaschine 242
MemoryDump-Verfahren 255
Messages 107
MIDI 146
MIDI-Befehle 156
MIDI-Befehlunterteilung 156
MIDI-Clock 249
MIDI-Codes 175
MIDI-Datenformat 155
MIDI-Device 175, 185
MIDI-Device-Strukturen 177
MIDI-Echtzeitbefehl 155
MIDI-Geräte 225
MIDI-Implementation-Chart 169
MIDI-Kanäle 151

Taken from Amiga Magazine Website

MIDI-Kanalnummer setzen – Event 275
MIDI-kompatible Geräte 220
MIDI-Modes 152
MIDI-Preset setzen – Event 275
MIDI-Schaltschemata 148
MIDI-Schnittstelle 145
MIDI-Spezifikation 156, 171
Midi.device 186
MIDIDemo 195
MIDIEvent-Typen 175
MIDIMon.asm 187
Mikrofon 130
Modulation 89, 100
Mono-Mode 154
MSB 33, 155
Multitrackrecording 237

N

Nachhall 139
Natürliche Periodizität 136
Nichtlineare Syntese 223
Non-Standard-Chunk 276
Note off 157
Note on 157
Note-Event 270
NoteDemo.c 320
Notenschlüssel setzen-Event 275
NTSC 63
Null-Abgleich 134
Numerische Werte 256

O

Oberwellen 22, 67
Omni-Mode 153
Omni-on/off-Schalter 152
One-Shot-Instrumente 136
Optionaler Text-Chunk 267 f.
Overdubbing 250

P

PAL 63
Paula 60, 70
Pause-Event 270
PCM-Synthese 221
Periodenzähler 74
Periodenzähler-Wert 64
Periodizität 26
Phase Distortion-Synthese 224
Phasengang 242
Phasensprung 137
Phasing 240
Pich Wheel 163
Ping-Pong 246
Pitch-Parameter 235
Play.asm 310
PlayNote 297, 307
PlayScore 294, 306
PlayTone 297, 308
Poly-Mode 153
Poly/Mono-Schalter 152
Ports 107
Program Change 163
Programm 187, 194, 216
Programmspezifische Chunks 276
PROP-Chunk 262
Properties 261
Prozessor-Betrieb 95
Punch In 246

Q

Quantisierung 32
Quantisierungsrauschen 41
Quantisierungsstufen 40
Quarter Frame 164

R

Rauschen 26
Rauschunterdrückungssysteme 244
Realtime-Events 176
Rechteckwelle 24
Rekonstruktionsfehler 52
Release 30
ReleaseTone 308
RLSE-Chunk 286
Roland U-220 233

Taken from Amiga-Manuals-Website

S

Sample&Hold-Verstärker 33, 39
Sample-Datenfeld 65
Sampling 16, 33, 129
Sampling-Rate 33, 133
Schlagzeug-Track 139
Schlagzeug.c 314
Schlagzeugmaschinen 230
Schwingung 20
Schwingungsdurchlauf 65
Schwingungsfunktionen 24
Score-Set 304
Sequencer 145
SetDefault 296, 307
SetPacker 301
SetReturnSig 298, 309
Shannons-Abtasttheorem 44
SHDR-Chunk 265
Sicherungskopie 324
Signal-/Rauschverhältnis 134
Signalquelle 130
Signedbinary-Verfahren 35
Slave 145
SMPTE-Timecode 248
SMUS-FORM 276
SMUS-Format 263
SMUS-IFF-Standard 263, 294
SMUS-Standardausdrücke 279
SMUSPlayer 293
SMUSPlayer-Developers-Kit 253
SMUSPlayer-Funktionen 294
SMUSPlayer-Library 293
Song Position Pointer 165
Song-Select-Befehl 166
Sonix 138
–, Dynamik 139
–, Nachhall 139
Speicherbedarf 64, 256
Spektraler Verlauf 27
Spektrum 21
Standard-8SVX-Chunk 282
Standard-Kommandos 181
Standard-SMUS-Chunk 265
Start 167
Status-Byte 155
Stereoverstärker 131
Sternschaltung 151
Stimmung 133
Stop 168, 306
Stop.asm 313
Störabstand 41
Störeinflüsse 132
String-Gadgets 303
Stroboskopeffekt 46
Subtraktive Synthese 222
Sustain 30
Sync-Track 249
SyncDemo.c 322
Synchroner IO 183
Synchronisation 247
Syntheseverfahren 221
Synthesizer 227, 229
Synthesizer-Slaves 233
System Common-Befehl 164
–, Exclusive Message 177
–, Exclusive-Befehl 164
–, Realtime-Befehl 164
–, Reset 170
System-Funktionen 179

T

Taktmaß setzen – Event 273
Tempo setzen – Event 275
Textdaten 256
Tiefpaßfilter 52
Timecode 248
Timing Clock 167
Tonart setzen – Event 274
Tonband 131
Töne 16
Tonhöhe 19
Tonhöhenkorrektur 135
Tonhöhenvariation 235
Touch-Pickups 130
Tracks 264
TRAK-Chunk 268

Taken from Amiga Stuff Website

Transiente Störspannung 54
Tremolo 89
Tune Request 166

U

Übergänge 137
Übersprech-Effekt 243
Übersteuerung 43
Übertragungsbereich 48
UnLoadScore 305
UnLoadSet 305
Unteraussteuerung 42

V

VHDR-Chunk 283
Vibrato 89, 235, 239
Video-Raster 63

W

Wah-Wah 240
Wandlerregister 96
Weißes Rauschen 27
Wellenform 25, 68
Wiedergabe 54
Wiedergaberate 58, 65

Y

Yamaha RX-8 231
Yamaha V50 229

Z

Zeitquantisierung 44
Zustandsübergänge 101
Zustandsübergangsdiagramm 100
Zweierkomplement 34

Taken from Amiga-Manuals-Website

AMIGA Audio Entwickler-Paket

Wer in seinem Amiga zum ersten Mal ein Spiel mit aufwendigem Sound lädt, glaubt meist, seinen Ohren nicht zu trauen: Aus dem Rechner, der keineswegs an eine Hi-Fi-Anlage erinnert, explodiert ein tonales Feuerwerk, und zwar mehrstimmig und in Stereo.

Nicolaus Wirsing, gleichermaßen Musik- wie Computerfachmann, hat dieses kleine Wunder zum Anlaß genommen, in seinem Kompendium all das zusammenzutragen, was man zum Thema Amiga und Musik wissen muß. Er erklärt in Teil 1, welche elektronischen Virtuosen für den guten Ton sorgen, welche Hard- und Software der Rechner dazu verwendet und wie man Musik, Sprache und Geräusche wie mit einem Kassettenrecorder aufnehmen und wiedergeben kann.

Verbindet man ein Keyboard oder eine Drum-Maschine mit dem Amiga, entsteht ein leistungsstarkes Musikstudio, mit dem man sowohl Rap-Musikern als auch Tanzorchestern Konkurrenz machen kann. Der Autor liefert alle Informationen über Midi, Keyboards und Aufzeichnungsverfahren.

In Teil 2 beschreibt Nicolaus Wirsing das »SMUSPlayer Developers Kit«, mit dem ohne großen Aufwand Musik in die selbstgeschriebenen Basic-, C- und Assemblerprogramme kommt. Eine SMUS-Player-Library, das Herzstück, befindet sich auf der beiliegenden Diskette. Digitalisierte Sprache etwa kann über die Library mit dem Programmablauf synchronisiert werden. Das Programm »SetPacker« erlaubt darüber hinaus, mehrere Soundpassagen gleichzeitig im Speicher zu halten und bei Bedarf abzuspielen. Demonstrationsprogramme, beispielsweise der CLI-Befehl »Play« für Sonix-Dateien, liegen bei.

Der Inhalt im Überblick:

- Wie Töne entstehen
- Die digitale Verarbeitung von Tonsignalen
- Vom Chip zum Ohr: wie der Amiga Sound erzeugt
- Systemsoftware: das Audio-Device
- Von der Idee zum Sound: samplen, Instrumente erstellen, komponieren
- Midi: Hardware, Software, Standards, Midi-Device, Midi-kompatible Instrumente
- Multitrack- und Masterrecording: Bandmaschinen, Synchronisation, Aufnahme
- Das »SMUSPlayer Developer Kit«: alles über das SMUS-IFF-Format, das Chunk-Prinzip, die Instrumenten-Formate
- Die SMUS-Player-Library: Programmierhilfe, das Programm SetPacker, die Funktionen der Library
- Die Demoprogramme

Hardware-Anforderungen:

Amiga mit mindestens 512 Kbyte Arbeitsspeicher. Empfehlenswert, aber nicht notwendig: Midi-Interface, Keyboard, Sound-Digitizer

Software-Anforderungen:

Empfehlenswert, aber nicht notwendig: ein Musikprogramm (etwa Deluxe Music oder Sonix), ein Sequencer-Programm bei Midi-Nutzung

ISBN 3-89090-765-2

09800

9 783890 907659

DM 98,– sFr 91,– öS 834,–

Unverbindliche Preisempfehlung

Markt&Technik

Archiv Commodore Amiga Website